Microsoft

Forth Edition

CLR via C#
(第4版)

（美）Jeffrey Richter 著
周靖 译



- 权威力作全新全面升级
- 名著名译再显经典魅力
- 根据Microsoft .NET Framework 4.5和Visual C# 2012全面更新
- 聚焦于Framework Class Library(FCL)的核心类型
- 清楚阐述多核编程、泛型、线程处理等基本概念

MVP
Microsoft Most Valuable Professional

联袂推荐

清华大学出版社

CLR via C#
(第4版)

MVP Microsoft Most Valuable Professional

联袂推荐

由浅入深，透视CLR和.NET精髓，知其然
鞭辟入里，探究CLR和.NET机理，知其所以然

要想深入探究并掌握CLR(公共语言运行时)、C#和.NET开发，其实很简单。

享有全球盛誉的编程专家Jeffrey Richter，微软.NET团队合作长达十几年的资深技术顾问，在本书中以简洁流畅的语言和读者分享他毕生的经验和心得，他的见解独到、深刻，他的深思熟虑和远见卓识，为开发人员构建健壮、可靠和响应快的应用程序与组件奠定了良好的基础。

《CLR via C#(第4版)》针对.NET Framework 4.5和多核编程进行了全面更新和修订，是帮助读者深入探索和掌握公共语言运行时、C#和.NET开发的重要参考。

本书重要主题

- 根据.NET Framework 4.5和Visual Studio 2012全面更新
- 深挖.NET Framwork架构、运行时环境和其他关键主题，包括异步编程和最新的Windows Runtime
- 提供丰富的Visual C# 2012范例代码
- 对泛型和线程处理等难以理解的概念提供实际有用的指导和讲解

➡ 本书使用的Visual C# 2012范例代码可以从作者网站下载：
http://wintellect.com/books

作者简介



Jeffrey Richter

Wintellect联合创始人。数十年如一日痴迷于Windows和.NET的大师，数十年以来影响了若干代程序员的灵魂人物，经典著作《Windows核心编程》和《CLR via C#》系列版本的缔造者。他崇尚大道至简，注重效率与实用性，尤其热爱化繁为简，去芜存菁。最值得一提的是，他特别擅长授人以渔，能够以通俗易懂、条理清晰的方式将普通读者望而却步的关键概念讲得透彻，讲得精彩。

Jeffrey出版过12本Windows/.NET畅销书，曾经担任过MSDN Magazine特约编辑、Win32 Q&A专栏作者、.NET Q&A专栏作者和Concurrent Affairs(关于并发那些事儿)专栏作者。他还在很多全球性的商业研讨会上发表演讲。作为最懂Windows和.NET的人，他从1990年开始就以顾问身份为微软提供服务，他写的代码先后被微软的很多产品广泛采用。此外，他还曾经为AT&T、IBM、英特尔、梦工厂、通用和惠普等提供过咨询服务。

Jeffrey爱好广泛，他拥有飞机驾照和直升机驾照，是国际魔法师协会成员。他爱好音乐(尤其是20世纪70年代的爵士乐和前卫摇滚乐)、击鼓、火车模型和空手道。他还喜欢旅游和戏剧。目前，他与爱妻Kristin及爱子Aidan和Grant居住在华盛顿州的柯克兰。

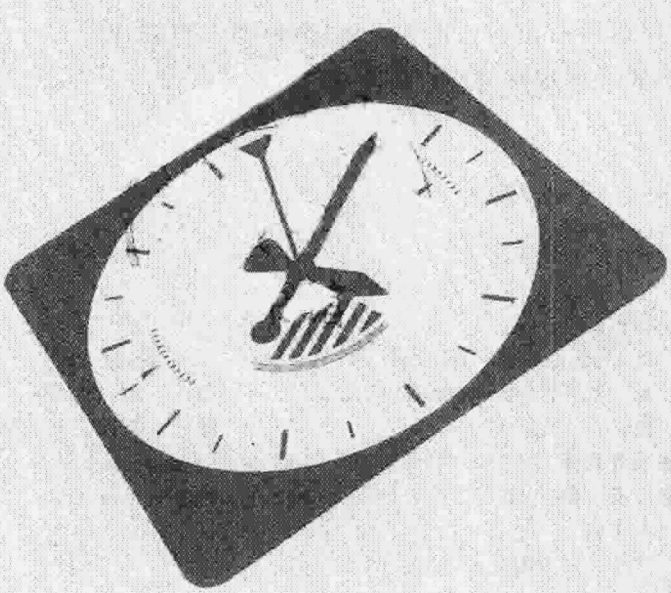
ISBN 978-7-302-38097-9
9 787302 380979 >
定价：109.00元

清华大学出版社数字出版网站
WQBook 书文局桌
www.wqbook.com

CLR via C#

(第4版)

（美）Jeffrey Richter 著
周 靖 译



清华大学出版社
北 京

内 容 简 介

本书针对 CLR 和.NET Framework 4.5 进行深入、全面的探讨，并结合实例介绍了如何利用它们进行设计、开发和调试。全书 5 部分共 29 章。第Ⅰ部分介绍 CLR 基础，第Ⅱ部分解释如何设计类型，第Ⅲ部分介绍基本类型，第Ⅳ部分以核心机制为主题，第Ⅴ部分重点介绍线程处理。

通过本书的阅读，读者可以掌握 CLR 和.NET Framework 的精髓，轻松、高效地创建高性能应用程序。

© 2014 Tsinghua University Press Limited
Authorized translation of the English edition of Microsoft CLR via C#, 4th Edition.
Copyright © 2012 by Jeffrey Richter. This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls of all rights to publish and sell the same.

本书中文版由 O'Reilly Media, Inc.授权给清华大学出版社出版发行，未经出版者许可，不得以任何方式复制或抄袭本书的任何部分。

北京市版权局著作权合同登记号　图字：01-2013-3404

版权所有，翻印必究。举报电话：010-62782989　13701121933
本书封面贴有清华大学出版社防伪标签，无标签者不得销售。

图书在版编目(CIP)数据

CLR via C#：第 4 版/(美)李希特(Richter, J.)著；周靖译. —北京：清华大学出版社，2015
ISBN 978-7-302-38097-9

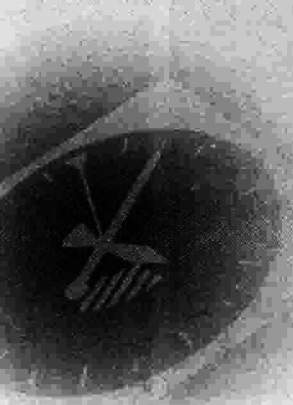
Ⅰ. ①C…　Ⅱ. ①李…　②周…　Ⅲ. C 语言—程序设计　Ⅳ. TP312

中国版本图书馆 CIP 数据核字(2014)第 221083 号

责任编辑：文开琪
装帧设计：杨玉兰
责任印制：宋　林

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　**邮　　编**：100084
社 总 机：010-62770175　　**邮　　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 刷 者：清华大学印刷厂
装 订 者：三河市新茂装订有限公司
经　　销：全国新华书店
开　　本：185mm×260mm　**印　　张**：46.75　**插　　页**：1　**字　　数**：1021 千字
版　　次：2003 年 11 月第 1 版　2015 年 1 月第 4 版　**印　　次**：2015 年 1 月第1 次印刷
印　　数：1～3500
定　　价：109.00 元

产品编号：050691-01



序　言

大家好，我们又见面了。谁预见到了今天啊？哈哈，我就预见到了！一旦步入婚姻的殿堂，就相当于过上了“土拨鼠日”。如果还没有看过那部电影，就去看看吧。看了之后，就会明白为什么自己老是犯同样的错误。当 Jeff 说他不再写书的时候，我就知道这是一个“瘾君子”开的空头支票。Jeff 不可能停止写书。就在今天，我们还在说起他“绝对”不可能写的另一本书呢(实际情况是，有一章已经在写了)。写书已深入到他骨子里面去了。千里马生来就是要奔跑的，Jeff 生来就是要写作的。

Jeff 太有规律了。他就是离不开硬盘里那些小小的 0 和 1。忽视它们是不可能的。凌晨 3 点，我们睡梦正酣的时候，Jeff 的生物钟就在催促他起床了(巧合的是，我们 4 岁大的小儿子也恰好在这个时候爬到我们的床上。爷儿俩的行为模式我都理解不了啊)。一股神秘的力量促使 Jeff 的大脑自动释放出解决方案、头脑风暴和臭虫之类的东西，迫使他跑到办公室把这些东西从脑袋里倒腾出来。而我们呢，则安心地翻个身继续呼呼大睡，知道 Jeff 会解决那些问题——就像一个神秘的网络超级英雄，防止线程又成为薄弱环节[①]。

但积累这些知识供自己使用，这对 Jeff 来说远远不够。好东西不该独享。所以必须把它们传播开来，必须把它们写下来。知识就像电波，有心人能接收得到。这就是他为你所做的，亲爱的读者，是他热爱微软技术的明证。

本书还有另一层意义在里面。Jeff 每次绕着太阳“公转”一圈，都会变得更老一些。经过多年的积累，他也学着“向后看”了。由于看事情的方式变得更成熟，所以他重写了讲反射的那一章。或许你也应该跟他一起回顾一下这个主题。可以学到怎样让代码自个儿询问关于代码的事儿，进而更深入地思考为什么反射要那样工作。穿上便服，找一把舒服的皮椅子坐下，花些时间想想自己的代码以及它们生命中更深层次的意义。

本书还讲了一样有趣的东西，就是异步/等待[②]。和我老公以前鼓捣过一阵子的 **AsyncEnumerator** 相比，这个东西显然进步了不少。哎，我还以为今后离不开它了呢！事实上，虽然他跟我讲了好多次 **AsyncEnumerator**，但这个东西根本就没有在我的脑子里“阻塞”嘛！于是我窃想，如果知道什么是 enumerator 的话，也许就能明白他讲的是啥了。于是我查了一下维基百科，发现 enumerator 是人口普查员的意思。这一章难道是讲人口普查

---

① saving the thread from becoming just another loose end，直译就是“防止线头儿又松了”。——译注

② **async** 和 **await** 是 C#的两个关键字，允许用顺序编程模型执行异步操作。——译注

员怎样协调工作的事儿？太浪费纳税人的钱了吧！不过，我相信它在计算机里面的意义比我查到的好。Jeff 和微软的团队一起工作，将异步/等待打磨得很完美。你现在通过这本书就能舒舒服服地享受他们的成果了。我建议你好好读一下。嗯，要顺着读。[①]

本书的另一个重头戏是我感觉最兴奋的。希望你们都来看看关于 WinRT 的内容。这个术语太书呆子气了，我的理解就是“马上为我无敌帅气的平板搞一些很酷的应用出来！”你猜得没错，新的 Windows Runtime 就是围绕无敌帅气的触摸屏展开的。孩子喜欢小鸟飞向小猪，我则喜欢跟鲜花有关的东西，而你完全可以用平板做其他事情。没有做不到，只有想不到！去折腾出一些“奇思妙想非常牛掰”(Wonderful Innovative Nifty Really Touchy, WinRT)的东西出来。就当是为了我，好好看看这一章。否则的话，我对 Jeff 和他无休止的写作事业可能真的会失去耐心，会把他关到一间只有针头线脑[②]而且没有电的小黑屋里面。你们程序员看着办吧，是用 WinRT 写一些很酷的应用，还是再也没有 Jeff 的新书看！

总之，在你们的力挺之下，Jeff 的又一部大作诞生了。我们的家庭貌似又可以回归正常状态了。但真的正常吗？或许他不停写书才是真的正常吧。

让我们耐心等待下一本书的神秘召唤。

Kristin Trace(Jeffrey 的妻子)

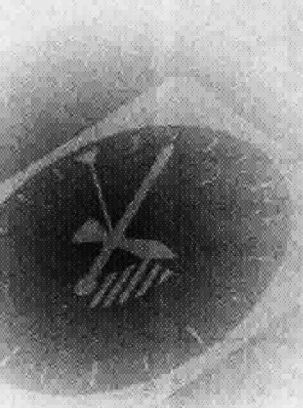
2012 年 10 月



救命！Jeff 要被关小黑屋了！

① Kristin 用“sequentially”一词来吐槽顺序编程模型。——译注

② Kristin 又在吐槽“thread”了。——译注



前　言

1999 年 10 月，Microsoft 的一些人首次向我展示了 Microsoft .NET Framework、公共语言运行时(CLR)和 C#编程语言。看到这一切时，我惊呆了，我知道我写软件的方式要发生非常大的变化了。他们请我为团队做一些顾问工作，我当即同意了。刚开始，我以为.NET Framework 是 Win32 API 和 COM 上的一个抽象层。但随着我投入越来越多的时间研究，我意识到它是一个更宏伟的项目。某种程度上，它是自己的操作系统。有自己的内存管理器，自己的安全系统，自己的文件加载器，自己的错误处理机制，自己的应用程序隔离边界(AppDomain)、自己的线程处理模型等。本书解释了所有这些主题，帮你为这个平台高效地设计和实现应用程序和组件。

我写这本书是 2012 年 10 月，距离首次接触.NET Framework 和 C#正好 13 年。13 年来，我以 Microsoft 顾问身份开发过各式各样的应用程序，为.NET Framework 本身也贡献良多。作为我自己公司 Wintellect(*http://Wintellect.com*)的合伙人，我还要为大量客户工作，帮他们设计、调试、优化软件以及解决使用.NET Framework 时遇到的问题。正是因为有了这些资历，所以我才知道如何用.NET Framework 进行高效率编程。贯穿本书所有主题，你都会看到我的经验之谈。

本书面向的读者

本书旨在解释如何为.NET Framework 开发应用程序和可重用的类。具体地说，我要解释 CLR 的工作原理及其提供的功能，还要讨论 Framework Class Library(FCL)的各个部分。没有一本书能完整地解释 FCL——其中含有数以千计的类型，而且这个数字正在以惊人速度增长。所以，我准备将重点放在每个开发人员都需要注意的核心类型上面。另外，虽然不会专门讲 Windows 窗体、Windows Presentation Foundation(WPF)、Microsoft Silverlight、XML Web 服务、Web 窗体、Microsoft ASP.NET MVC、Windows Store 应用等，但本书描述的技术适用于所有这些应用程序类型。

本书围绕 Microsoft Visual Studio 2012/2013，.NET Framework 4.5.*x* 和 C# 5.0 展开。由于 Microsoft 在发布这些技术的新版本时，会试图保持很大程度的向后兼容性，所以本书描述的许多内容也适合之前的版本。所有示例代码都用 C#编程语言写成。但由于 CLR 可由许多编程语言使用，所以本书内容也适合非 C#程序员。



> 注意　本书代码可从 Wintellect 网站下载：*http://wintellect.com/Resource-CLR-Via-CSharp-Fourth-Edition*，也可从译者博客下载：*http://transbot.blog.163.com*。

我和我的编辑进行了艰苦卓绝的工作，试图为你提供最准确、最新、最深入、最容易阅读和理解、没有错误的信息。但是，即便有如此完美的团队协作，疏漏和错误也在所难免。如果你发现了本书的任何错误或者想提出一些建设性的意见，请发送邮件到 *JeffreyR@Wintellect.com*。

致谢

没有别人的帮助和技术援助，我是不可能写好这本书的。尤其要感谢我的家人。写好一本书所投入的时间和精力无法衡量。我只知道，没有我的妻子 Kristin 和两个儿子 Aidan 和 Grant 的支持，根本不可能有这本书的面世。多少次想花些时间一家人小聚，都因为本书而放弃。现在，本书总算告一段落，终于有时间做自己喜欢做的事情了。

本书的修订得到了一些“高人”的协助。.NET Framework 团体队的一些人(其中许多都是我的朋友)审阅了其中的章节，我和他们进行了许多发人深省的对话。Christophe Nasarre 参与了我几本书的出版，在审阅本书并确保我能以最恰当的方式来表达的过程中，表现出了非凡的才能。他对本书的品质有至关重要的影响。和往常一样，我和 Microsoft Press 的团队进行了令人愉快的合作。特别感谢 Ben Ryan，Devon Musgrave 和 Carol Dillingham。另外，感谢 Susie Carr 和 Candace Sinclair 提供的编辑和制作支持。

勘误和支持

我们尽最大努力保证本书的准确性。勘误或更改会添加到以下网页：

http://www.oreilly.com/catalog/errata.csp?isbn=0790145353665

http://go.microsoft.com/FWLink/?Linkid=266601

如果发现未列出的错误，可通过相同的网页报告。

如需其他支持，请致函 Microsoft Press Book Support 部门：

mspinput@microsoft.com

注意，上述邮件地址不提供产品支持。

最后，本书中文版的支持(勘误和资源下载)请访问译者博客：

http://transbot.blog.163.com

简明目录

第Ⅰ部分 CLR基础

第1章 CLR的执行模型 3
第2章 生成、打包、部署和管理应用程序及类型 29
第3章 共享程序集和强命名程序集 57

第Ⅱ部分 设计类型

第4章 类型基础 81
第5章 基元类型、引用类型 和值类型 99
第6章 类型和成员基础 135
第7章 常量和字段 155
第8章 方法 161
第9章 参数 185
第10章 属性 201
第11章 事件 221
第12章 泛型 233
第13章 接口 259

第Ⅲ部分 基本类型

第14章 字符、字符串和文本处理 279
第15章 枚举类型和位标志 319
第16章 数组 329
第17章 委托 345

第 18 章　定制特性 371

第 19 章　可空值类型 389

第Ⅳ部分　核心机制

第 20 章　异常和状态管理 399

第 21 章　托管堆和垃圾回收 447

第 22 章　CLR 寄宿和 AppDomain 489

第 23 章　程序集加载和反射 515

第 24 章　运行时序列化 541

第 25 章　与 WinRT 组件互操作 569

第Ⅴ部分　线程处理

第 26 章　线程基础 591

第 27 章　计算限制的异步操作 611

第 28 章　I/O 限制的异步操作 643

第 29 章　基元线程同步构造 669

第 30 章　混合线程同步构造 697

目　录

第 I 部分　CLR 基础

第 1 章　CLR 的执行模型 3

1.1　将源代码编译成托管模块 3
1.2　将托管模块合并成程序集 6
1.3　加载公共语言运行时 7
1.4　执行程序集的代码 10
1.4.1　IL 和验证 15
1.4.2　不安全的代码 16
1.5　本机代码生成器：NGen.exe 17
1.6　Framework 类库 20
1.7　通用类型系统 22
1.8　公共语言规范 24
1.9　与非托管代码的互操作性 28

第 2 章　生成、打包、部署和管理应用程序及类型 29

2.1　.NET Framework 部署目标 29
2.2　将类型生成到模块中 31
2.3　元数据概述 33
2.4　将模块合并成程序集 39
2.4.1　使用 Visual Studio IDE 将程序集添加到项目中 44
2.4.2　使用程序集链接器 45
2.4.3　为程序集添加资源文件 46
2.5　程序集版本资源信息 47
2.6　语言文化 51
2.7　简单应用程序部署(私有部署的程序集) 52
2.8　简单管理控制(配置) 53

第 3 章　共享程序集和强命名程序集 57

3.1　两种程序集，两种部署 58
3.2　为程序集分配强名称 59
3.3　全局程序集缓存 63
3.4　在生成的程序集中引用强命名程序集 65
3.5　强命名程序集能防篡改 66
3.6　延迟签名 67
3.7　私有部署强命名程序集 69
3.8　“运行时”如何解析类型引用 70
3.9　高级管理控制(配置) 73

第 II 部分　设计类型

第 4 章　类型基础 81

4.1　所有类型都从 System.Object 派生 81
4.2　类型转换 83
4.3　命名空间和程序集 86
4.4　运行时的相互关系 90

第 5 章　基元类型、引用类型和值类型 99

5.1　编程语言的基元类型 99

5.2 引用类型和值类型 106
5.3 值类型的装箱和拆箱 111
5.3.1 使用接口更改已装箱值类型中的字段(以及为什么不应该这样做) 121
5.3.2 对象相等性和同一性 123
5.4 对象哈希码 126
5.5 dynamic 基元类型 127

第 6 章 类型和成员基础 135

6.1 类型的各种成员 135
6.2 类型的可见性 138
6.3 成员的可访问性 139
6.4 静态类 141
6.5 分部类、结构和接口 142
6.6 组件、多态和版本控制 143
6.6.1 CLR 如何调用虚方法、属性和事件 145
6.6.2 合理使用类型的可见性和成员的可访问性 148
6.6.3 对类型进行版本控制时的虚方法的处理 150

第 7 章 常量和字段 155

7.1 常量 155
7.2 字段 156

第 8 章 方法 161

8.1 实例构造器和类(引用类型) 161
8.2 实例构造器和结构(值类型) 164
8.3 类型构造器 167
8.4 操作符重载方法 170
8.5 转换操作符方法 173
8.6 扩展方法 176
8.6.1 规则和原则 178
8.6.2 用扩展方法扩展各种类型 179
8.6.3 ExtensionAttribute 类 181
8.7 分部方法 181

第 9 章 参数 185

9.1 可选参数和命名参数 185
9.1.1 规则和原则 186
9.1.2 DefaultParameterValueAttribute 和 OptionalAttribute 188
9.2 隐式类型的局部变量 188
9.3 以传引用的方式向方法传递参数 190
9.4 向方法传递可变数量的参数 195
9.5 参数和返回类型的设计规范 197
9.6 常量性 198

第 10 章 属性 201

10.1 无参属性 201
10.1.1 自动实现的属性 204
10.1.2 合理定义属性 205
10.1.3 对象和集合初始化器 208
10.1.4 匿名类型 209
10.1.5 System.Tuple 类型 212
10.2 有参属性 214
10.3 调用属性访问器方法时的性能 218
10.4 属性访问器的可访问性 219
10.5 泛型属性访问器方法 219

第 11 章 事件 221

11.1 设计要公开事件的类型 222
11.1.1 第一步：定义类型来容纳所有需要发送给事件通知接收者的附加信息 222
11.1.2 第二步：定义事件成员 223
11.1.3 第三步：定义负责引发事件的方法来通知事件的登记对象 224
11.1.4 第四步：定义方法将输入转化为期望事件 226
11.2 编译器如何实现事件 226
11.3 设计侦听事件的类型 228
11.4 显式实现事件 230

第 12 章　泛型.....233

12.1　FCL 中的泛型.....237
12.2　泛型基础结构.....238
12.2.1　开放类型和封闭类型.....239
12.2.2　泛型类型和继承.....240
12.2.3　泛型类型同一性.....242
12.2.4　代码爆炸.....243
12.3　泛型接口.....243
12.4　泛型委托.....244
12.5　委托和接口的逆变和协变泛型类型实参.....245
12.6　泛型方法.....247
12.7　泛型和其他成员.....249
12.8　可验证性和约束.....250
12.8.1　主要约束.....252
12.8.2　次要约束.....253
12.8.3　构造器约束.....254
12.8.4　其他可验证性问题.....254

第 13 章　接口.....259

13.1　类和接口继承.....259
13.2　定义接口.....260
13.3　继承接口.....261
13.4　关于调用接口方法的更多探讨.....263
13.5　隐式和显式接口方法实现(幕后发生的事情).....264
13.6　泛型接口.....266
13.7　泛型和接口约束.....268
13.8　实现多个具有相同方法名和签名的接口.....269
13.9　用显式接口方法实现来增强编译时类型安全性.....270
13.10　谨慎使用显式接口方法实现.....271
13.11　设计：基类还是接口.....274

第Ⅲ部分　基本类型

第 14 章　字符、字符串和文本处理.....279

14.1　字符.....279
14.2　System.String 类型.....282
14.2.1　构造字符串.....282
14.2.2　字符串是不可变的.....284
14.2.3　比较字符串.....285
14.2.4　字符串留用.....290
14.2.5　字符串池.....293
14.2.6　检查字符串中的字符和文本元素.....293
14.2.7　其他字符串操作.....295
14.3　高效率构造字符串.....296
14.3.1　构造 StringBuilder 对象.....296
14.3.2　StringBuilder 的成员.....297
14.4　获取对象的字符串表示：ToString.....299
14.4.1　指定具体的格式和语言文化.....299
14.4.2　将多个对象格式化成一个字符串.....303
14.4.3　提供定制格式化器.....304
14.5　解析字符串来获取对象：Parse.....306
14.6　编码：字符和字节的相互转换.....308
14.6.1　字符和字节流的编码和解码.....313
14.6.2　Base-64 字符串编码和解码.....314
14.7　安全字符串.....315

第 15 章　枚举类型和位标志.....319

15.1　枚举类型.....319
15.2　位标志.....324
15.3　向枚举类型添加方法.....328

第 16 章　数组.....329

16.1　初始化数组元素.....331
16.2　数组转型.....333

16.3 所有数组都隐式派生自 System.Array 335
16.4 所有数组都隐式实现 IEnumerable, ICollection 和 IList 336
16.5 数组的传递和返回 337
16.6 创建下限非零的数组 338
16.7 数组的内部工作原理 339
16.8 不安全的数组访问和固定大小的数组 342

第 17 章 委托 345

17.1 初识委托 345
17.2 用委托回调静态方法 347
17.3 用委托回调实例方法 349
17.4 委托揭秘 349
17.5 用委托回调多个方法(委托链) 353
17.5.1 C#对委托链的支持 356
17.5.2 取得对委托链调用的更多控制 357
17.6 委托定义不要太多(泛型委托) 359
17.7 C#为委托提供的简化语法 360
17.7.1 简化语法 1：不需要构造委托对象 360
17.7.2 简化语法 2：不需要定义回调方法(lambda 表达式) 361
17.7.3 简化语法 3：局部变量不需要手动包装到类中即可传给回调方法 364
17.8 委托和反射 367

第 18 章 定制特性 371

18.1 使用定制特性 371
18.2 定义自己的特性类 374
18.3 特性构造器和字段/属性数据类型 377
18.4 检测定制特性 378
18.5 两个特性实例的相互匹配 382
18.6 检测定制特性时不创建从 Attribute 派生的对象 384
18.7 条件特性类 387

第 19 章 可空值类型 389

19.1 C#对可空值类型的支持 391
19.2 C#的空接合操作符 393
19.3 CLR 对可空值类型的特殊支持 394
19.3.1 可空值类型的装箱 394
19.3.2 可空值类型的拆箱 395
19.3.3 通过可空值类型调用 GetType 395
19.3.4 通过可空值类型调用接口方法 395

第Ⅳ部分 核 心 机 制

第 20 章 异常和状态管理 399

20.1 定义“异常” 399
20.2 异常处理机制 401
20.2.1 try 块 402
20.2.2 catch 块 402
20.2.3 finally 块 404
20.3 System.Exception 类 407
20.4 FCL 定义的异常类 410
20.5 抛出异常 412
20.6 定义自己的异常类 413
20.7 用可靠性换取开发效率 415
20.8 设计规范和最佳实践 422
20.8.1 善用 finally 块 423
20.8.2 不要什么都捕捉 424
20.8.3 得体地从异常中恢复 425
20.8.4 发生不可恢复的异常时回滚部分完成的操作——维持状态 426
20.8.5 隐藏实现细节来维系协定 427
20.9 未处理的异常 429
20.10 对异常进行调试 433
20.11 异常处理的性能问题 435
20.12 约束执行区域(CER) 438

20.13 代码协定 441

第 21 章 托管堆和垃圾回收 447

21.1 托管堆基础 447
21.1.1 从托管堆分配资源 448
21.1.2 垃圾回收算法 449
21.1.3 垃圾回收和调试 451
21.2 代：提升性能 454
21.2.1 垃圾回收触发条件 458
21.2.2 大对象 459
21.2.3 垃圾回收模式 459
21.2.4 强制垃圾回收 462
21.2.5 监视应用程序的内存使用 463
21.3 使用需要特殊清理的类型 464
21.3.1 使用包装了本机资源的类型 470
21.3.2 一个有趣的依赖性问题 474
21.3.3 GC 为本机资源提供的其他功能 475
21.3.4 终结的内部工作原理 479
21.3.5 手动监视和控制对象的生存期 481

第 22 章 CLR 寄宿和 AppDomain 489

22.1 CLR 寄宿 489
22.2 AppDomain 491
跨越 AppDomain 边界访问对象 494
22.3 卸载 AppDomain 504
22.4 监视 AppDomain 505
22.5 AppDomain FirstChance 异常通知 507
22.6 宿主如何使用 AppDomain 507
22.6.1 可执行应用程序 507
22.6.2 Microsoft Silverlight 富 Internet 应用程序 508
22.6.3 Microsoft ASP.NET 和 XML Web 服务应用程序 508
22.6.4 Microsoft SQL Server 509
22.6.5 更多的用法只局限于想象力 509
22.7 高级宿主控制 509
22.7.1 使用托管代码管理 CLR 509
22.7.2 写健壮的宿主应用程序 510
22.7.3 宿主如何拿回它的线程 511

第 23 章 程序集加载和反射 515

23.1 程序集加载 516
23.2 使用反射构建动态可扩展应用程序 520
23.3 反射的性能 521
23.3.1 发现程序集中定义的类型 522
23.3.2 类型对象的准确含义 522
23.3.3 构建 Exception 派生类型的层次结构 524
23.3.4 构造类型的实例 525
23.4 设计支持加载项的应用程序 527
23.5 使用反射发现类型的成员 529
23.5.1 发现类型的成员 530
23.5.2 调用类型的成员 533
23.5.3 使用绑定句柄减少进程的内存消耗 537

第 24 章 运行时序列化 541

24.1 序列化/反序列化快速入门 542
24.2 使类型可序列化 546
24.3 控制序列化和反序列化 548
24.4 格式化器如何序列化类型实例 551
24.5 控制序列化/反序列化的数据 552
24.6 流上下文 558
24.7 类型序列化为不同类型以及对象反序列化为不同对象 559
24.8 序列化代理 562
24.9 反序列化对象时重写程序集/类型 566

第 25 章 与 WinRT 组件互操作 569

25.1 CLR 投射与 WinRT 组件类型系统规则 571

25.2 框架投射 575
25.2.1 从.NET 代码中调用异步 WinRT API 575
25.2.2 WInRT 流和.NET 流之间的互操作 579
25.2.3 在 CLR 和 WinRT 之间传输数据块 580
25.3 用 C#定义 WinRT 组件 583

第V部分 线 程 处 理

第 26 章 线程基础 591

26.1 Windows 为什么要支持线程 591
26.2 线程开销 592
26.3 停止疯狂 595
26.4 CPU 发展趋势 597
26.5 CLR 线程和 Windows 线程 598
26.6 使用专用线程执行异步的计算限制操作 599
26.7 使用线程的理由 601
26.8 线程调度和优先级 603
26.9 前台线程和后台线程 608
26.10 继续学习 609

第 27 章 计算限制的异步操作 611

27.1 CLR 线程池基础 612
27.2 执行简单的计算限制操作 612
27.3 执行上下文 614
27.4 协作式取消和超时 615
27.5 任务 619
27.5.1 等待任务完成并获取结果 620
27.5.2 取消任务 622
27.5.3 任务完成时自动启动新任务 623
27.5.4 任务可以启动子任务 625
27.5.5 任务内部揭秘 625
27.5.6 任务工厂 627
27.5.7 任务调度器 628
27.6 Parallel 的静态 For，ForEach 和 Invoke 方法 630
27.7 并行语言集成查询(PLINQ) 634
27.8 执行定时计算限制操作 636
27.9 线程池如何管理线程 639
27.9.1 设置线程池限制 639
27.9.2 如何管理工作者线程 640

第 28 章 I/O 限制的异步操作 643

28.1 Windows 如何执行 I/O 操作 643
28.2 C#的异步函数 647
28.3 编译器如何将异步函数转换成状态机 649
28.4 异步函数扩展性 653
28.5 异步函数和事件处理程序 655
28.6 FCL 的异步函数 656
28.7 异步函数和异常处理 658
28.8 异步函数的其他功能 658
28.9 应用程序及其线程处理模型 661
28.10 以异步方式实现服务器 663
28.11 取消 I/O 操作 664
28.12 有的 I/O 操作必须同步进行 665
28.13 I/O 请求优先级 666

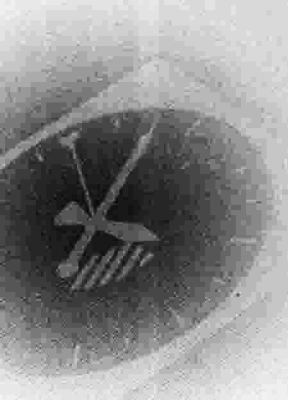
第 29 章 基元线程同步构造 669

29.1 类库和线程安全 671
29.2 基元用户模式和内核模式构造 672
29.3 用户模式构造 673
29.3.1 易变构造 674
29.3.2 互锁构造 678
29.3.3 实现简单的自旋锁 682
29.3.4 Interlocked Anything 模式 685
29.4 内核模式构造 687
29.4.1 Event 构造 691
29.4.2 Semaphore 构造 693
29.4.3 Mutex 构造 694

第 30 章　混合线程同步构造.....697

30.1　一个简单的混合锁.....697
30.2　自旋、线程所有权和递归.....699
30.3　FCL 中的混合构造.....701
30.3.1　ManualResetEventSlim 类和 SemaphoreSlim 类.....701
30.3.2　Monitor 类和同步块.....701
30.3.3　ReaderWriterLockSlim 类.....706
30.3.4　OneManyLock 类.....709
30.3.5　CountdownEvent 类.....711
30.3.6　Barrier 类.....711
30.3.7　线程同步构造小结.....712
30.4　著名的双检锁技术.....713
30.5　条件变量模式.....717
30.6　异步的同步构造.....719
30.7　并发集合类.....723

译者后记.....727



第 I 部分　CLR 基础

- 第 1 章　CLR 的执行模型
- 第 2 章　生成、打包、部署和管理应用程序及类型
- 第 3 章　共享程序集和强命名程序集

第3章 [illegible]
第5章 [illegible]
第4章 CPU的指令系统

第1部分 CPU原理

第 1 章　CLR 的执行模型

本章内容：

- 将源代码编译成托管模块
- 将托管模块合并成程序集
- 加载公共语言运行时
- 执行程序集的代码
- 本机代码生成器：NGen.exe
- Framework 类库入门
- 通用类型系统
- 公共语言规范(CLS)
- 与非托管代码的互操作性

Microsoft .NET Framework 引入了许多新概念、技术和术语。本章概述了.NET Framework 如何设计，介绍了 Framework 包含的一些新技术，并定义了今后要用到的许多术语。还要展示如何将源代码生成为一个应用程序，或者生成为一组可重新分发的组件(文件)——这些组件(文件)中包含类型(类和结构等)。最后，本章解释了应用程序如何执行。

1.1　将源代码编译成托管模块

决定将.NET Framework 作为自己的开发平台之后，第一步便是决定要生成什么类型的应用程序或组件。假定你已完成了这个小细节；一切均已设计好，规范已经写好，可以着手开发了。

现在，必须决定要使用哪一种编程语言。这通常是一个艰难的抉择，因为不同的语言各有长处。例如，非托管 C/C++可对系统进行低级控制。可完全按自己的想法管理内存，必要时能方便地创建线程。另一方面，使用 Microsoft Visual Basic 6.0 可以快速生成 UI 应用程序，并可方便地控制 COM 对象和数据库。

顾名思义，**公共语言运行时**(Common Language Runtime，CLR)是一个可由多种编程语言使用的“运行时”。[①]CLR 的核心功能(比如内存管理、程序集加载、安全性、异常处理和线

① CLR 在早期文档中翻译为“公共语言运行库”，但“库”一词很容易让人误解，所以本书翻译为“公共语言运行时”，或简称为“运行时”。为了和“程序运行的时候”区分，“运行时”在作为名词时会添加引号。——译注

程同步)可由面向 CLR 的所有语言使用。例如，“运行时”使用异常来报告错误；因此，面向它的任何语言都能通过异常来报告错误。另外，“运行时”允许创建线程，所以面向它的任何语言都能创建线程。

事实上，在运行时，CLR 根本不关心开发人员用哪一种语言写源代码。这意味着在选择编程语言时，应选择最容易表达自己意图的语言。可用任何编程语言开发代码，只要编译器是面向 CLR 的。

既然如此，不同编程语言的优势何在呢？事实上，可将编译器视为语法检查器和“正确代码”分析器。它们检查源代码，确定你写的一切都有意义，并输出对你的意图进行描述的代码。不同编程语言允许用不同的语法来开发。不要低估这个选择的价值。例如，对于数学或金融应用程序，使用 APL 语法来表达自己的意图，相较于使用 Perl 语法来表达同样的意图，可以节省许多开发时间。

Microsoft 创建好了几个面向“运行时”的语言编译器，其中包括：C++/CLI、C#(发音是“C sharp”)、Visual Basic、F#(发音是“F sharp”)、Iron Python、Iron Ruby 以及一个“中间语言”(Intermediate Language，IL)汇编器。除了 Microsoft，另一些公司、学院和大学也创建了自己的编译器，也能面向 CLR 生成代码。我所知道的有针对下列语言的编译器：Ada，APL，Caml，COBOL，Eiffel，Forth，Fortran，Haskell，Lexico，LISP，LOGO，Lua，Mercury，ML，Mondrian，Oberon，Pascal，Perl，PHP，Prolog，RPG，Scheme，Smalltalk 和 Tcl/Tk。

图 1-1 展示了编译源代码文件的过程。如图所示，可用支持 CLR 的任何语言创建源代码文件，然后用对应的编译器检查语法和分析源代码。无论选择哪个编译器，结果都是**托管模块**(managed module)。托管模块是标准的 32 位 Microsoft Windows 可移植执行体(PE32)文件[①]，或者是标准的 64 位 Windows 可移植执行体(PE32+)文件，它们都需要 CLR 才能执行。顺便说一句，托管程序集总是利用 Windows 的数据执行保护(Data Execution Prevention，DEP)和地址空间布局随机化(Address Space Layout Randomization，ASLR)，这两个功能旨在增强整个系统的安全性。

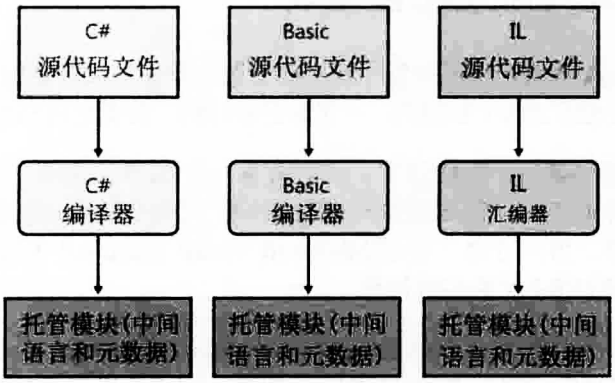


图 1-1　将源代码编译成托管模块

① PE 是 Portable Executable(可移植执行体)的简称。——译注

表 1-1 总结了托管模块的各个组成部分。

表 1-1　托管模块的各个部分

组成部分	说明
PE32 或 PE32+头	标准 Windows PE 文件头，类似于“公共对象文件格式”(Common Object File Format，COFF)头。如果这个头使用 PE32 格式，文件能在 Windows 的 32 位或 64 位版本上运行。如果这个头使用 PE32+格式，文件只能在 Windows 的 64 位版本上运行。这个头还标识了文件类型，包括 GUI，CUI 或者 DLL，并包含一个时间标记来指出文件的生成时间。对于只包含 IL 代码的模块，PE32(+)头的大多数信息会被忽视。如果是包含本机(native)CPU 代码的模块，这个头包含与本机 CPU 代码有关的信息
CLR 头	包含使这个模块成为托管模块的信息(可由 CLR 和一些实用程序进行解释)。头中包含要求的 CLR 版本，一些标志(flag)，托管模块入口方法(Main 方法)的 MethodDef 元数据 token 以及模块的元数据、资源、强名称、一些标志及其他不太重要的数据项的位置/大小
元数据	每个托管模块都包含元数据表。主要有两种表：一种表描述源代码中定义的类型和成员，另一种描述源代码引用的类型和成员
IL(中间语言)代码	编译器编译源代码时生成的代码。在运行时，CLR 将 IL 编译成本机 CPU 指令

本机代码编译器(native code compilers)生成的是面向特定 CPU 架构(比如 x86，x64 或 ARM)的代码。相反，每个面向 CLR 的编译器生成的都是 IL(中间语言)代码。(本章稍后会详细讨论 IL 代码。)IL 代码有时称为**托管代码**(managed code)，因为 CLR 管理它的执行。

除了生成 IL，面向 CLR 的每个编译器还要在每个托管模块中生成完整的元数据(metadata)。元数据简单地说就是一个数据表集合。一些数据表描述了模块中定义了什么(比如类型及其成员)，另一些描述了模块引用了什么(比如导入的类型及其成员)。元数据是一些老技术的超集。这些老技术包括 COM 的“类型库”(Type Library)和“接口定义语言”(Interface Definition Language，IDL)文件。但 CLR 元数据远比它们全面。另外，和类型库及 IDL 不同，元数据总是与包含 IL 代码的文件关联。事实上，元数据总是嵌入和代码相同的 EXE/DLL 文件中，这使两者密不可分。由于编译器同时生成元数据和代码，把它们绑定一起，并嵌入最终生成的托管模块，所以元数据和它描述的 IL 代码永远不会失去同步。

元数据有多种用途，下面仅列举一部分。

- 元数据避免了编译时对原生 C/C++头和库文件的需求，因为在实现类型/成员的 IL 代码文件中，已包含有关引用类型/成员的全部信息。编译器直接从托管模块读取元数据。
- Microsoft Visual Studio 用元数据帮助你写代码。“智能感知”(IntelliSense)技术会解析元数据，告诉你一个类型提供了哪些方法、属性、事件和字段。对于方法，还能告诉你需要的参数。

- CLR 的代码验证过程使用元数据确保代码只执行“类型安全”的操作。(稍后就会讲到验证。)
- 元数据允许将对象的字段序列化到内存块，将其发送给另一台机器，然后反序列化，在远程机器上重建对象状态。
- 元数据允许垃圾回收器跟踪对象生存期。垃圾回收器能判断任何对象的类型，并从元数据知道那个对象中的哪些字段引用了其他对象。

第 2 章“生成、打包、部署和管理应用程序及类型”将更详细地讲述元数据。

Microsoft 的 C#，Visual Basic，F#和 IL 汇编器总是生成包含托管代码(IL)和托管数据(可进行垃圾回收的数据类型)的模块。为了执行包含托管代码以及/或者托管数据的模块，最终用户必须在自己的计算机上安装好 CLR(目前作为.NET Framework 的一部分提供)。这类似于为了运行 MFC 或者 Visual Basic 6.0 应用程序，用户必须安装 Microsoft Foundation Class(MFC)库或者 Visual Basic DLL。

Microsoft 的 C++编译器默认生成包含非托管(native)代码的 EXE/DLL 模块，并在运行时操纵非托管数据(native 内存)。这些模块不需要 CLR 即可执行。然而，通过指定**/CLR** 命令行开关，C++编译器就能生成包含托管代码的模块。当然，最终用户必须安装 CLR 才能执行这种代码。在前面提到的所有 Microsoft 编译器中，C++编译器是独一无二的，只有它才允许开发人员同时写托管和非托管代码，并生成到同一个模块中。它也是唯一允许开发人员在源代码中同时定义托管和非托管数据类型的 Microsoft 编译器。Microsoft C++编译器的灵活性是其他编译器无法比拟的，因为它允许开发人员在托管代码中使用原生 C/C++代码，时机成熟后再使用托管类型。

1.2 将托管模块合并成程序集

CLR 实际不和模块工作。它和程序集工作。**程序集**(assembly)是抽象概念，初学者很难把握它的精髓。首先，程序集是一个或多个模块/资源文件的逻辑性分组。其次，程序集是重用、安全性以及版本控制的最小单元。取决于你选择的编译器或工具，既可生成单文件程序集，也可生成多文件程序集。在 CLR 的世界中，程序集相当于“组件”。

第 2 章会深入探讨程序集，这里不打算花费太多笔墨。只想提醒一句：利用“程序集”这种概念性的东西，一组文件可作为一个单独的实体来对待。

图 1-2 有助于你理解程序集。图中一些托管模块和资源(或数据)文件准备交由一个工具处理。工具生成代表文件逻辑分组的一个 PE32(+)文件。实际发生的事情是，这个 PE32(+)文件包含一个名为**清单**(manifest)的数据块。清单也是元数据表的集合。这些表描述了构成程序集的文件、程序集中的文件所实现的公开导出的类型[①]以及与程序集关联的资源或数据文件。

① 所谓公开导出的类型，就是程序集中定义的 public 类型，它们在程序集内部外部均可见。——译注

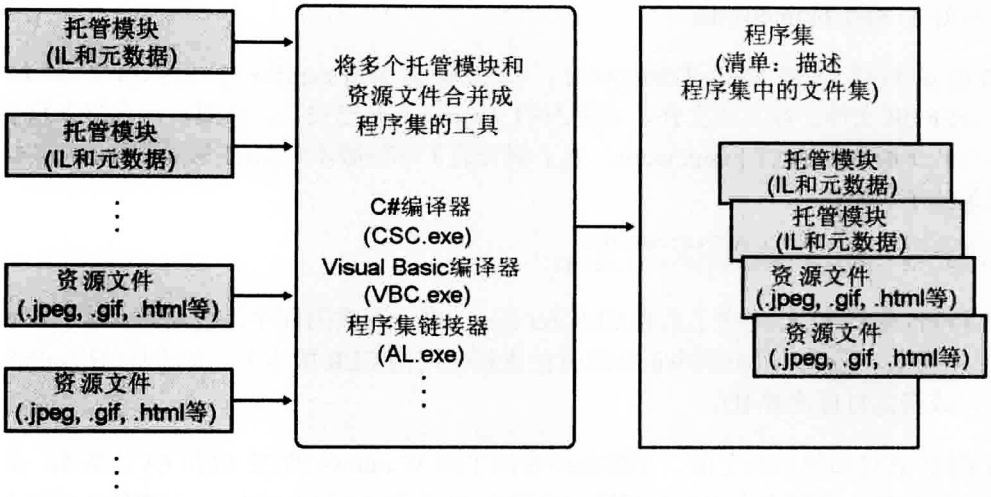


图 1-2　将托管模块合并成程序集

编译器默认将生成的托管模块转换成程序集。也就是说，C#编译器生成的是含有清单的托管模块。清单指出程序集只由一个文件构成。所以，对于只有一个托管模块而且无资源(或数据)文件的项目，程序集就是托管模块，生成过程中无需执行任何额外的步骤。但是，如果希望将一组文件合并到程序集中，就必须掌握更多的工具(比如程序集链接器 AL.exe)及其命令行选项。第 2 章将解释这些工具和选项。

对于一个可重用的、可保护的、可版本控制的组件，程序集把它的逻辑表示和物理表示区分开。具体如何用不同的文件划分代码和资源，这完全取决于个人。例如，可以将很少用到的类型或资源放到单独的文件中，并把这些文件作为程序集的一部分。取决于运行时的需要，可从网上下载这些单独的文件。如果文件永远用不上，则永远不会下载。这样不仅节省磁盘空间，还缩短了安装时间。利用程序集，可以在不同的地方部署文件，同时仍然将所有文件作为一个整体来对待。

在程序集的模块中，还包含与引用的程序集有关的信息(包括它们的版本号)。这些信息使程序集能够**自描述**(self-describing)。也就是说，CLR 能判断为了执行程序集中的代码，程序集的直接依赖对象(immediate dependency)是什么。不需要在注册表或 Active Directory Domain Services(ADDS)中保存额外的信息。由于无需额外信息，所以和非托管组件相比，程序集更容易部署。

1.3　加载公共语言运行时

生成的每个程序集既可以是可执行应用程序，也可以是 DLL(其中含有一组由可执行程序使用的类型)。当然，最终是由 CLR 管理这些程序集中的代码的执行。这意味着目标机器必须安装好.NET Framework。Microsoft 创建了一个重分发包(redistribution package)，允许将.NET Framework 免费分发并安装到用户的计算机上。一些版本的 Windows 在发售时就已

经打包好了.NET Framework。

要知道是否已安装.NET Framework，只需检查%SystemRoot%\System32 目录中的MSCorEE.dll 文件。存在该文件，表明.NET Framework 已安装。然而，一台机器可能同时安装好几个版本的.NET Framework。要了解安装了哪些版本的.NET Framework，请检查以下目录的子目录：

```
%SystemRoot%\Microsoft.NET\Framework
%SystemRoot%\Microsoft.NET\Framework64
```

.NET Framework SDK 提供了名为 CLRVer.exe 的命令行实用程序，能列出机器上安装的所有 CLR 版本。还能列出机器中正在运行的进程使用的 CLR 版本号，方法是使用**-all** 命令行开关，或指定目标进程 ID。

学习 CLR 具体如何加载之前，稍微花些时间了解 Windows 的 32 位和 64 位版本。如果程序集文件只包含类型安全的托管代码，代码在 32 位和 64 位 Windows 上都能正常工作。在这两种 Windows 上运行，源代码无需任何改动。事实上，编译器最终生成的 EXE/DLL 文件在 Windows 的 x86 和 x64 版本上都能正常工作。此外，Windows Store 应用或类库能在 Windows RT 机器(使用 ARM CPU)上运行。也就是说，只要机器上安装了对应版本的.NET Framework，文件就能运行。

极少数情况下，开发人员希望代码只在一个特定版本的 Windows 上运行。例如，要使用不安全的代码，或者要和面向一种特定 CPU 架构的非托管代码进行互操作，就可能需要这样做。为了帮助这些开发人员，C#编译器提供了一个**/platform** 命令行开关选项。这个开关允许指定最终生成的程序集只能在运行 32 位 Windows 版本的 x86 机器上使用，只能在运行 64 位 Windows 的 x64 机器上使用，或者只能在运行 32 位 Windows RT 的 ARM 机器上使用。不指定具体平台的话，默认选项就是 **anycpu**，表明最终生成的程序集能在任何版本的 Windows 上运行。Visual Studio 用户要想设置目标平台，可以打开项目的属性页，从“生成”选项卡的“目标平台”列表中选择一个选项，如图 1-3 所示。

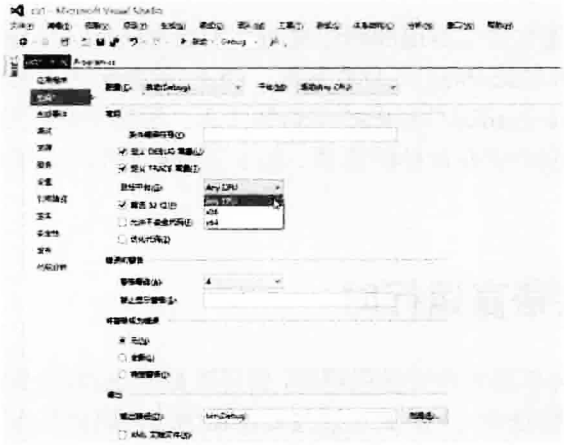


图 1-3 使用 Visual Studio 设置目标平台

取决于**/platform** 开关选项，C#编译器生成的程序集包含的要么是 PE32 头，要么是 PE32+头。除此之外，编译器还会在头中指定要求什么 CPU 架构(如果使用默认值 **anycpu**，则代表任意 CPU 架构)。Microsoft 发布了 SDK 命令行实用程序 DumpBin.exe 和 CorFlags.exe，可用它们检查编译器生成的托管模块所嵌入的信息。

可执行文件运行时，Windows 检查文件头，判断需要 32 位还是 64 位地址空间。PE32 文件在 32 位或 64 位地址空间中均可运行，PE32+文件则需要 64 位地址空间。Windows 还会检查头中嵌入的 CPU 架构信息，确保当前计算机的 CPU 符合要求。最后，Windows 的 64 位版本通过 WoW64(Windows on Windows64)技术运行 32 位 Windows 应用程序。

表 1-2 总结了两方面的信息。其一，为 C#编译器指定不同**/platform** 命令行开关将得到哪种托管模块。其二，应用程序在不同版本的 Windows 上如何运行。

表 1-2　/platform 开关选项对生成的模块的影响以及在运行时的影响

/platform 开关	生成的托管模块	x86 Windows	x64 Windows	ARM Windows RT
anycpu(默认)	PE32/任意 CPU 架构	作为 32 位应用程序运行	作为 64 位应用程序运行	作为 32 位应用程序运行
anycpu32bitpreferred	PE32/任意 CPU 架构	作为 32 位应用程序运行	作为 WoW64 应用程序运行	作为 32 位应用程序运行
x86	PE32/x86	作为 32 位应用程序运行	作为 WoW64 应用程序运行	不运行
x64	PE32+/x64	不运行	作为 64 位应用程序运行	不运行
ARM	PE32/ARM	不运行	不运行	作为 32 位应用程序运行

Windows 检查 EXE 文件头，决定是创建 32 位还是 64 位进程之后，会在进程地址空间加载 MSCorEE.dll 的 x86，x64 或 ARM 版本。如果是 Windows 的 x86 或 ARM 版本，MSCorEE.dll 的 x86 版本在%SystemRoot%\System32 目录中。如果是 Windows 的 x64 版本，MSCorEE.dll 的 x86 版本在%SystemRoot%\SysWow64 目录中，64 位版本则在%SystemRoot%\System32 目录中(为了向后兼容)。然后，进程的主线程调用 MSCorEE.dll 中定义的一个方法。这个方法初始化 CLR，加载 EXE 程序集，再调用其入口方法(**Main**)。随即，托管应用程序启动并运行。[①]

① 可在代码中查询 **Environment** 的 **Is64BitOperatingSystem** 属性，判断是否在 64 位 Windows 上运行。还可查询 **Environment** 的 **Is64BitProcess** 属性，判断是否在 64 位地址空间中运行。



注意 Microsoft C#编译器 1.0 或 1.1 版本生成的程序集包含的是 PE32 头，而且未明确指定 CPU 架构。但在加载时，CLR 认为这些程序集只用于 x86。对于可执行文件，这增强了应用程序与 64 位系统的兼容能力，因为可执行文件将在 WoW64 中加载，为进程提供和 Windows 的 32 位 x86 版本非常相似的环境。

如果非托管应用程序调用 **LoadLibrary** 加载托管程序集，Windows 会自动加载并初始化 CLR(如果尚未加载)以处理程序集中的代码。当然，这个时候进程已经启动并运行了，而这可能限制程序集的可用性。例如，64 位进程完全无法加载使用**/platform:x86** 开关编译的托管程序集，而用相同开关编译的可执行文件能在 64 位 Windows 中用 WoW64 进行加载。

1.4 执行程序集的代码

如前所述，托管程序集同时包含元数据和 IL。IL 是与 CPU 无关的机器语言，是 Microsoft 在请教了外面的几个商业及学术性语言/编译器的作者之后，费尽心思开发出来的。IL 比大多数 CPU 机器语言都高级。IL 能访问和操作对象类型，并提供了指令来创建和初始化对象、调用对象上的虚方法以及直接操作数组元素。甚至提供了抛出和捕捉异常的指令来实现错误处理。可将 IL 视为一种面向对象的机器语言。

开发人员一般用 C#，Visual Basic 或 F#等高级语言进行编程。它们的编译器将生成 IL。然而，和其他任何机器语言一样，IL 也能使用汇编语言编写，Microsoft 甚至专门提供了名为 ILAsm.exe 的 IL 汇编器和名为 ILDasm.exe 的 IL 反汇编器。

注意，高级语言通常只公开了 CLR 全部功能的一个子集。然而，IL 汇编语言允许开发人员访问 CLR 的全部功能。所以，如果你选择的编程语言隐藏了你迫切需要的一个 CLR 功能，可以换用 IL 汇编语言或者提供了所需功能的另一种编程语言来写那部分代码。



重要提示 在我看来，允许在不同编程语言之间方便地切换，同时又保持紧密集成，这是 CLR 的一个很出众的特点。遗憾的是，许多开发人员都忽视了这一点。例如，C#和 Visual Basic 等语言能很好地执行 I/O 操作，APL 语言能很好地执行高级工程或金融计算。通过 CLR，应用程序的 I/O 部分可用 C#编写，工程计算部分则换用 APL 编写。CLR 在这些语言之间提供了其他技术无法媲美的集成度，使“混合语言编程”成为许多开发项目一个值得慎重考虑的选择。

要知道 CLR 具体提供了哪些功能，唯一的办法就是阅读 CLR 文档。本书致力于讲解 CLR 的功能，以及 C#语言如何公开这些功能。对于 C#没有公开的 CLR 功能，本书也进行了说明。相比之下，其他大多数书籍和文章都是从一种语言的角度讲解 CLR，造成大多数开发

人员误以为 CLR 只提供了他们选用的那一种语言所公开的那一部分功能。不过，只要用一种语言就能达到目的，这种误解不一定是坏事。

为了执行方法，首先必须把方法的 IL 转换成本机(native)CPU 指令。这是 CLR 的 JIT(just-in-time 或者“即时”)编译器的职责。

图 1-4 展示了一个方法首次调用时发生的事情。

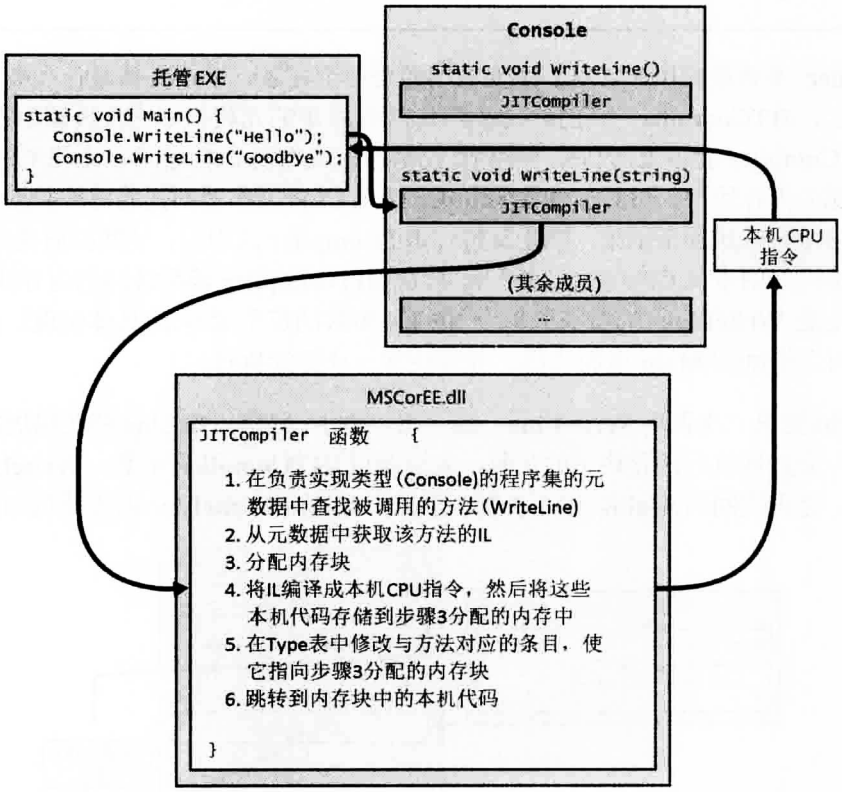


图 1-4　方法的首次调用

就在 **Main** 方法执行之前，CLR 会检测出 **Main** 的代码引用的所有类型。这导致 CLR 分配一个内部数据结构来管理对引用类型的访问。图 1-4 的 **Main** 方法引用了一个 **Console** 类型，导致 CLR 分配一个内部结构。在这个内部数据结构中，**Console** 类型定义的每个方法都有一个对应的记录项[①]。每个记录项都含有一个地址，根据此地址即可找到方法的实现。对这个结构初始化时，CLR 将每个记录项都设置成(指向)包含在 CLR 内部的一个未编档函数。我将该函数称为 **JITCompiler**。

Main 方法首次调用 **WriteLine** 时，**JITCompiler** 函数会被调用。**JITCompiler** 函数负责将

① 本书将 entry 翻译成“记录项”，其他译法还有条目、入口等等。虽然某些 entry 包含了一个地址，所以相当于一个指针，但并非所有 entry 都是这样的。在其他 entry 中，还可能包含了文件名、类型名、方法名和位标志等信息。——译注

方法的 IL 代码编译成本机 CPU 指令。由于 IL 是“即时”(just in time)编译的，所以通常将 CLR 的这个组件称为 JITter 或者 JIT **编译器**。



> **注意** 如果应用程序在 Windows 的 x86 版本或 WoW64 中运行，JIT 编译器将生成 x86 指令。作为 64 位应用程序在 Windows 的 x64 版本中运行，将生成 x64 指令。在 Windows 的 ARM 版本中运行，将生成 ARM 指令。

JITCompiler 函数被调用时，它知道要调用的是哪个方法，以及具体是什么类型定义了该方法。然后，**JITCompiler** 会在定义(该类型的)程序集的元数据中查找被调用方法的 IL。接着，**JITCompiler** 验证 IL 代码，并将 IL 代码编译成本机 CPU 指令。本机 CPU 指令保存到动态分配的内存块中。然后，**JITCompiler** 回到 CLR 为类型创建的内部数据结构，找到与被调用方法对应的那条记录，修改最初对 **JITCompiler** 的引用，使其指向内存块(其中包含了刚才编译好的本机 CPU 指令)的地址。最后，**JITCompiler** 函数跳转到内存块中的代码。这些代码正是 **WriteLine** 方法(获取单个 **String** 参数的那个版本)的具体实现。代码执行完毕并返回时，会回到 **Main** 中的代码，并像往常一样继续执行。

现在，**Main** 要第二次调用 **WriteLine**。这一次，由于已对 **WriteLine** 的代码进行了验证和编译，所以会直接执行内存块中的代码，完全跳过 **JITCompiler** 函数。**WriteLine** 方法执行完毕后，会再次回到 **Main**。图 1-5 展示了第二次调用 **WriteLine** 时发生的事情。

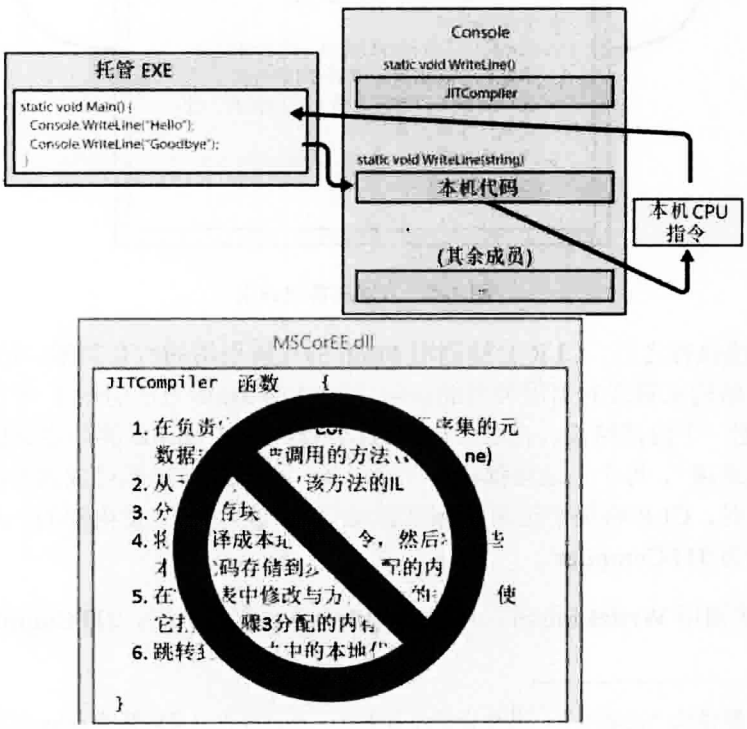


图 1-5 方法的第二次调用

方法仅在首次调用时才会有一些性能损失。以后对该方法的所有调用都以本机代码的形式全速运行，无需重新验证 IL 并把它编译成本机代码。

JIT 编译器将本机 CPU 指令存储到动态内存中。这意味着一旦应用程序终止，编译好的代码也会被丢弃。所以，将来再次运行应用程序，或者同时启动应用程序的两个实例(使用两个不同的操作系统进程)，JIT 编译器必须再次将 IL 编译成本机指令。对于某些应用程序，这可能显著增加内存耗用。相比之下，本机(native)应用程序的只读代码页可由应用程序正在运行的所有实例共享。

对于大多数应用程序，JIT 编译造成的性能损失并不显著。大多数应用程序都反复调用相同的方法。应用程序运行期间，这些方法只会对性能造成一次性的影响。另外，在方法内部花费的时间很有可能比花在调用方法上的时间多得多。

还要注意，CLR 的 JIT 编译器会对本机代码进行优化，这类似于非托管 C++编译器的后端所做的事情。同样，可能花较多时间生成优化代码。但和没有优化时相比，代码优化后性能更佳。

编译器开关设置	C# IL 代码质量	JIT 本机代码质量
/optimize- /debug-(默认)	未优化	有优化
/optimize- /debug(+/full/pdbonly)	未优化	未优化
/optimize+ /debug(-/+/full/pdbonly)	有优化	有优化

两个 C#编译器开关会影响代码优化：**/optimize** 和**/debug**。下面总结了这些开关对 C#编译器生成的 IL 代码的质量的影响，以及对 JIT 编译器生成的本机代码的质量的影响。

使用**/optimize-**，在 C#编译器生成的未优化 IL 代码中，将包含许多 NOP(no-operation，空操作)指令，还包含许多跳转到下一行代码的分支指令。Visual Studio 利用这些指令在调试期间提供“编辑并继续”(edit-and-continue)功能。另外，利用这些额外的指令，还可在控制流程指令(比如 **for**，**while**，**do**，**if**，**else**，**try**，**catch** 和 **finally** 语句块)上设置断点，使代码更容易调试。相反，如果生成优化的 IL 代码，C#编译器会删除多余的 NOP 和分支指令。而在控制流程被优化之后，代码就难以在调试器中进行单步调试了。另外，若在调试器中执行，一些函数求值可能无法进行。不过，优化的 IL 代码变得更小，结果 EXE/DLL 文件也更小。另外，如果你像我一样喜欢检查 IL 来理解编译器生成的东西，这种 IL 更易读。

此外，只有指定**/debug(+/full/pdbonly)**开关，编译器才会生成 Program Database(PDB)文件。PDB 文件帮助调试器查找局部变量并将 IL 指令映射到源代码。**/debug:full** 开关告诉 JIT 编译器你打算调试程序集，JIT 编译器会记录每条 IL 指令所生成的本机代码。这样一来，就可利用 Visual Studio 的“即时”调试功能，将调试器连接到正在运行的进程，并方便地对源代码进行调试。不指定**/debug:full** 开关，JIT 编译器默认不记录 IL 与本机代码的联系，这使 JIT 编译器运行得稍快，用的内存也稍少。如果进程用 Visual Studio 调试器启动，会强迫 JIT 编译器记录 IL 与本机代码的联系(无论**/debug** 开关的设置是什么)——除非在 Visual Studio 中关闭了“在模块加载时取消 JIT 优化(仅限托管)”选项。

在 Visual Studio 中新建 C#项目时，项目的“调试”(Debug)配置指定的是**/optimize-**和**/debug:full** 开关，而“发布”(Release)配置指定的是**/optimize+**和**/debug:pdbonly** 开关。

非托管 C 或 C++的开发人员可能担心所有这些对于性能的影响。毕竟，非托管代码是针对一种具体 CPU 平台编译的。一旦调用，代码直接就能执行。但在现在这种托管环境中，代码的编译是分两个阶段完成的。首先，编译器遍历源代码，做大量工作来生成 IL 代码。但真正要想执行，这些 IL 代码本身必须在运行时编译成本机 CPU 指令，这需要分配更多的非共享内存，并要花费额外的 CPU 时间。

事实上，我自己也是从 C/C++的背景开始接触 CLR 的，当时也对此持怀疑态度并格外关心这种额外的开销。经过实践，我发现运行时的二次编译确实会影响性能，也确实会分配动态内存。但是，Microsoft 进行了大量性能优化工作，将这些额外的开销保持在最低限度之内。

如果仍不放心，就实际生成一些应用程序，亲自测试一下性能。此外，应该运行由 Microsoft 或其他公司生成的一些比较正式的托管应用程序，并测试其性能。相信它们出色的性能表现会让你喜出望外。

虽然你可能很难相信，但许多人(包括我)都认为托管应用程序的性能实际上超越了非托管应用程序。有许多原因使我们相信。例如，当 JIT 编译器在运行时将 IL 代码编译成本机代码时，编译器对执行环境的认识比非托管编译器更深刻。下面列举了托管代码相较于非托管代码的优势。

- JIT 编译器能判断应用程序是否运行在 Intel Pentium 4 CPU 上，并生成相应的本机代码来利用 Pentium 4 支持的任何特殊指令。相反，非托管应用程序通常是针对具有最小功能集合的 CPU 编译的，不会使用能提升性能的特殊指令。

- JIT 编译器能判断一个特定的测试在它运行的机器上是否总是失败。例如，假定一个方法包含以下代码：

  ```
  if (numberOfCPUs > 1) {
    ...
  }
  ```

  如果主机只有一个 CPU，JIT 编译器不会为上述代码生成任何 CPU 指令。在这种情况下，本机代码将针对主机进行优化，最终代码变得更小，执行得更快。

- 应用程序运行时，CLR 可以评估代码的执行，并将 IL 重新编译成本机代码。重新编译的代码可以重新组织，根据刚才观察到的执行模式，减少不正确的分支预测。虽然目前版本的 CLR 还不能做到这一点，但将来的版本也许就可以了。

除了这些理由，还有另一些理由使我们相信未来的托管代码在执行效率上会比当前的非托管代码更优秀。大多数托管应用程序目前的性能已相当不错，将来还有望进一步提升。

如果试验表明，CLR 的 JIT 编译器似乎没有使自己的应用程序达到应有的性能，那么为了进一步确认，还应该使用.NET Framework SDK 配套提供的 NGen.exe 工具。该工具将程序

集的所有 IL 代码编译成本机代码，并将这些本机代码保存到一个磁盘文件中。在运行时加载程序集时，CLR 自动判断是否存在该程序集的预编译版本。如果是，CLR 就加载预编译代码。这样一来，就避免了在运行时进行编译。注意，NGen.exe 对最终执行环境的预设是很保守的(不得不如此)。所以，NGen.exe 生成的代码不会像 JIT 编译器生成的代码那样进行高度优化。本章稍后将详细讨论 NGen.exe。

另外可以考虑使用 **System.Runtime.ProfileOptimization** 类。该类导致 CLR 检查程序运行时哪些方法被 JIT 编译，结果被记录到一个文件。程序再次启动时，如果是在多 CPU 机器上运行，就用其他线程并发编译这些方法。这使应用程序运行得更快，因为多个方法并发编译，而且是在应用程序初始化时编译，而不是在用户和程序交互时才“即时”编译。

1.4.1　IL 和验证

IL 基于栈。这意味着它的所有指令都要将操作数压入(push)一个执行栈，并从栈弹出(pop)结果。由于 IL 没有提供操作寄存器的指令，所以人们可以很容易地创建新的语言和编译器，生成面向 CLR 的代码。

IL 指令还是“无类型”(typeless)的。例如，IL 提供了 **add** 指令将压入栈的最后两个操作数加到一起。**add** 指令不分 32 位和 64 位版本。**add** 指令执行时，它判断栈中的操作数的类型，并执行恰当的操作。

我个人认为，IL 最大的优势不是它对底层 CPU 的抽象，而是应用程序的健壮性[①]和安全性。将 IL 编译成本机 CPU 指令时，CLR 执行一个名为**验证**(verification)的过程。这个过程会检查高级 IL 代码，确定代码所做的一切都是安全的。例如，会核实调用的每个方法都有正确数量的参数，传给每个方法的每个参数都有正确的类型，每个方法的返回值都得到了正确的使用，每个方法都有一个返回语句，等等。托管模块的元数据包含验证过程要用到的所有方法及类型信息。

Windows 的每个进程都有自己的虚拟地址空间。独立地址空间之所以必要，是因为不能简单地信任一个应用程序的代码。应用程序完全可能读写无效的内存地址(令人遗憾的是，这种情况时有发生)。将每个 Windows 进程都放到独立的地址空间，将获得健壮性与稳定性：一个进程干扰不到另一个进程。

然而，通过验证托管代码，可确保代码不会不正确地访问内存，不会干扰到另一个应用程序的代码。这样就可以放心地将多个托管应用程序放到同一个 Windows 虚拟地址空间运行。

由于 Windows 进程需要大量操作系统资源，所以进程数量太多，会损害性能并制约可用的

① 这里有必要强调一下健壮性(鲁棒性)和可靠性的区别，两者对应的英文单词分别是 robustness 和 reliability。健壮性主要描述系统对于参数变化的不敏感性，而可靠性主要描述系统的正确性，也就是在你固定提供一个参数时，它应该产生稳定的、能预测的输出。例如一个程序，它的设计目标是获取输入并输出值。假如它能正确完成这个设计目标，就说它是可靠的。但在这个程序执行完毕后，假如没有正确释放内存，或者说系统没有自动帮它释放占用的资源，就认为这个程序及其“运行时”不健壮。——译注

资源。用一个进程运行多个应用程序，可减少进程数，从而增强性能，减少所需的资源，健壮性也没有丝毫下降。这是托管代码相较于非托管代码的另一个优势。

事实上，CLR 确实提供了在一个操作系统进程中执行多个托管应用程序的能力。每个托管应用程序都在一个 AppDomain[①]中执行。每个托管 EXE 文件默认都在它自己的独立地址空间中运行，这个地址空间只有一个 AppDomain。然而，CLR 的宿主进程(比如 IIS 或者 Microsoft SQL Server)可决定在一个进程中运行多个 AppDomain。第 22 章“CLR 寄宿和 AppDomain”会详细讨论 AppDomain。

1.4.2 不安全的代码

Microsoft C#编译器默认生成安全(safe)代码，这种代码的安全性可以验证。然而，Microsoft C#编译器也允许开发人员写不安全的(unsafe)代码。不安全的代码允许直接操作内存地址，并可操作这些地址处的字节。这是非常强大的一个功能，通常只有在与非托管代码进行互操作，或者在提升对效率要求极高的一个算法的性能的时候，才需要这样做。

然而，使用不安全的代码存在重大风险：这种代码可能破坏数据结构，危害安全性，甚至造成新的安全漏洞。有鉴于此，C#编译器要求包含不安全代码的所有方法都用 **unsafe** 关键字标记。除此之外，C#编译器要求使用**/unsafe** 编译器开关来编译源代码。

当 JIT 编译器编译一个 **unsafe** 方法时，会检查该方法所在的程序集是否被授予了 **System.Security.Permissions.SecurityPermission** 权限，而且 **System.Security.Permissions.SecurityPermissionFlag** 的 **SkipVerification** 标志是否设置。如果该标志已经设置，JIT 编译器会编译不安全的代码，并允许代码执行。CLR 信任这些代码，并希望对地址及字节的直接操作不会造成损害。如果标志未设置，JIT 编译器会抛出 **System.InvalidProgramException** 或 **System.Security.VerificationException** 异常，禁止方法执行。事实上，整个应用程序都有可能在这个时候终止，但这至少能防止造成损害。



注意 从本地计算机或“网络共享”加载的程序集默认被授予完全信任，这意味着它们能做任何事情，包括执行不安全代码。但通过 Internet 执行的程序集默认不会被授予执行不安全代码的权限。如果含有不安全的代码，就会抛出上述异常之一。管理员和最终用户可以修改这些默认设置；但在这种情况下，管理员要对代码的行为负全责。

Microsoft 提供了一个名为 PEVerify.exe 的实用程序，它检查一个程序集的所有方法，并报告其中含有不安全代码的方法。对想要引用的程序集运行一下 PEVerify.exe，看看应用程序在通过内网或 Internet 运行时是否会出问题。

① 本书按照原书的风格保持了 AppDomain 这样的写法，未将其翻译成“应用程序域”。需引用 **AppDomain** 类的时候，会将其加粗显示。平时引用时，则采用普通字体。——译注

注意，验证要访问所有依赖的程序集中包含的元数据。所以，当 PEVerify 检查程序集时，它必须能够定位并加载引用的所有程序集。由于 PEVerify 使用 CLR 来定位依赖的程序集，所以会采用和平时执行程序集时一样的绑定(binding)和探测(probing)规则来定位程序集。这些绑定和探测规则将在第 2 章和第 3 章讨论。

IL 和知识产权保护

有的人担心 IL 没有为他们的算法提供足够的知识产权保护。换言之，他们认为在生成托管模块后，别人可以使用工具(比如 IL 反汇编器)来进行逆向工程，轻松还原应用程序的代码所做的事情。

我承认，IL 代码确实比其他大多数汇编语言高级，而且对 IL 代码进行逆向工程相对而言比较简单。不过，在实现服务器端代码(比如 Web 服务、Web 窗体或者存储过程)的时候，程序集是放在服务器上的。由于没人能从公司外部拿到程序集，所以没人能从公司外部使用工具查看 IL。所以，这个时候的知识产权是完全安全的。

如果担心分发出去的程序集，可从第三方厂商购买混淆器(obfuscator)实用程序。这种实用程序能打乱程序集元数据中的所有私有符号的名称。别人很难还原这些名称，从而很难理解每个方法的作用。但要注意，这些混淆器提供的保护是有限的，因为 IL 必须在某个时候提供给 CLR 进行 JIT 编译。

如果觉得混淆器不能提供自己需要的知识产权保护等级，可考虑在非托管模块中实现你想保密的算法。这种模块将包含本机 CPU 指令，而不是 IL 和元数据。然后，可利用 CLR 的互操作功能(假定有足够的权限)来实现应用程序的托管与非托管部分之间的通信。当然，上述方案的前提是不担心别人对非托管代码中的本机 CPU 指令进行逆向工程。

1.5　本机代码生成器：NGen.exe

使用.NET Framework 提供的 NGen.exe 工具，可以在应用程序安装到用户的计算机上时，将 IL 代码编译成本机代码。由于代码在安装时已经编译好，所以 CLR 的 JIT 编译器不需要在运行时编译 IL 代码，这有助于提升应用程序的性能。NGen.exe 能在以下两种情况下发挥重要作用。

1. **提高应用程序的启动速度**
   运行 NGen.exe 能提高启动速度，因为代码已编译成本机代码，运行时不需要再花时间编译。

2. **减小应用程序的工作集**[①]

① 所谓工作集(working set)，是指在进程的所有内存中，已映射的物理内存那一部分(即这些内存块全在物理内存中，并且 CPU 可以直接访问)；进程还有一部分虚拟内存，它们可能在转换列表中(CPU 不能通过虚地址访问，需要 Windows 映射之后才能访问)；还有一部分内存在磁盘上的分页文件里。——译注

如果一个程序集同时加载到多个进程中，对该程序集运行 NGen.exe 可减小应用程序的工作集。NGen.exe 将 IL 编译成本机代码，并将这些代码保存到单独的文件中。该文件可以通过“内存映射”的方式，同时映射到多个进程地址空间中，使代码得到了共享，避免每个进程都需要一份单独的代码拷贝。

安装程序为应用程序或程序集调用 NGen.exe 时，应用程序的所有程序集(或者那个指定的程序集)的 IL 代码会编译成本机代码。NGen.exe 新建一个程序集文件，其中只包含这种本机代码，不含任何 IL。新文件会放到%SystemRoot%\Assembly\NativeImages_v4.0.#####_64 这样的一个目录下的一个文件夹中。目录名称除了包含 CLR 版本号，还会描述本机代码是为 32 位还是 64 位 Windows 编译的。

现在，每当 CLR 加载程序集文件，都会检查是否存在一个对应的、由 NGen 生成的本机文件。如果找不到本机文件，CLR 就和往常一样对 IL 代码进行 JIT 编译。如果有对应的本机文件，CLR 就直接使用本机文件中编译好的代码，文件中的方法不需要在运行时编译。

表面上很完美！一方面，获得了托管代码的所有好处(垃圾回收、验证、类型安全等等)；另一方面，没有托管代码(JIT 编译)的所有性能问题。但是，不要被表面所迷惑。NGen 生成的文件有以下问题。

- **没有知识产权保护**

  许多人以为发布 NGen 生成的文件(而不发布包含原始 IL 代码的文件)能保护知识产权。但遗憾的是，这是不可能的。在运行时，CLR 要求访问程序集的元数据(用于反射和序列化等功能)，这就要求发布包含 IL 和元数据的程序集。此外，如果 CLR 因为某些原因不能使用 NGen 生成的文件(如后文所述)，CLR 会自动对程序集的 IL 代码进行 JIT 编译，所以 IL 代码必须处于可用状态。

- **NGen 生成的文件可能失去同步**

  CLR 加载 NGen 生成的文件时，会将预编译代码的许多特征与当前执行环境进行比较。任何特征不匹配，NGen 生成的文件就不能使用。此时要改为使用正常的 JIT 编译器进程。下面列举了必须匹配的部分特征。

  - CLR 版本：随补丁或 Service Pack 改变
  - CPU 类型：升级处理器发生改变
  - Windows 操作系统版本：安装新 Service Pack 后改变
  - 程序集的标识模块版本 ID(Module Version ID，MVID)：重新编译后改变
  - 引用的程序集的版本 ID：重新编译引用的程序集后改变
  - 安全性：吊销了之前授予的权限之后，安全性就会发生改变。这些权限包括声明

性继承(declarative inheritance)、声明性链接时(declarative link-time)[①]、**SkipVerification** 或者 **UnmanagedCode** 权限。



> **注意**　可以用更新(update)模式运行 NGen.exe，为以前用 NGen 生成的所有程序集再次运行 NGen.exe。用户一旦安装了.NET Framework 的新 Service Pack，这个 Service Pack 的安装程序就会自动用更新模式运行 NGen.exe，使 NGen 生成的文件与新安装的 CLR 版本同步。

- **较差的执行时性能**

  编译代码时，NGen 无法像 JIT 编译器那样对执行环境进行许多假定。这会造成 NGen.exe 生成较差的代码。例如，NGen 不能优化地使用特定 CPU 指令；静态字段只能间接访问，而不能直接访问，因为静态字段的实际地址只能在运行时确定。NGen 到处插入代码来调用类构造器，因为它不知道代码的执行顺序，也不知道一个类构造器是否已经调用。(第 8 章“方法”会详细讲述类构造器的问题)测试表明，相较于 JIT 编译的版本，NGen 生成的某些应用程序在执行时反而要慢 5%左右。所以，假如考虑使用 NGen.exe 来提升应用程序的性能，必须仔细比较 NGen 版本和非 NGen 版本，确定 NGen 版本不会变得更慢！对于某些应用程序，由于缩小工作集能提升性能，所以使用 NGen 仍有优势。

正是由于这些问题，所以使用 NGen.exe 时必须谨慎。对于服务器端应用程序，NGen.exe 的作用并不明显，有时甚至毫无用处，这是因为只有第一个客户端请求才会感受到性能下降，后续所有客户端请求都能以全速运行。此外，大多数服务器应用程序只需要代码的一个实例，所以缩小工作集不能带来任何好处。

对于客户端应用程序，使用 NGen.exe 也许能提高启动速度，或者能缩小工作集(如果程序集同时由多个应用程序使用)。即便程序集不由多个应用程序使用，用 NGen 来生成也可能会增强工作集。此外，用 NGen.exe 生成客户端应用程序的所有程序集，CLR 就不需要加载 JIT 编译器了，从而进一步缩小工作集。当然，只要有一个程序集不是用 NGen 生成的，或者程序集的一个由 NGen 生成的文件无法使用，那么还是会加载 JIT 编译器，应用程序的工作集将随之增大。

对于启动很慢的大型客户端应用程序，Microsoft 提供了 Managed Profile Guided Optimization 工具(MPGO.exe)。该工具分析程序执行，检查它在启动时需要哪些东西。这些信息反馈给 NGen.exe 来更好地优化本机映像，这使应用程序启动得更快，工作集也缩小了。准备发布应用程序时，用 MPGO 工具启动它，走一遍程序的常规任务。与所执行代码有关的信息会写入一个 profile 并嵌入程序集文件中。NGen.exe 工具利用这些 profile 数据来更好地优化它生成的本机映像。

---

① declarative inheritance 权限是派生出程序集的那个类所要求的；declarative link-time 权限是程序集调用的方法所要求的。另外，虽然文档将 declarative 翻译成“声明性”，但个人更喜欢“宣告式”。——译注

1.6 Framework 类库

.NET Framework 包含 Framework 类库(Framework Class Library，FCL)。FCL 是一组 DLL 程序集的统称，其中含有数千个类型定义，每个类型都公开了一些功能。Microsoft 还发布了其他库，比如 Windows Azure SDK 和 DirectX SDK。这些库提供了更多类型，公开了更多功能。事实上，Microsoft 正在以惊人的速度发布各种各样的库，开发者使用各种 Microsoft 技术变得前所未有的简单。

下面列举了应用程序开发人员可以利用这些程序集创建的一部分应用程序。

- **Web 服务(Web service)**
  利用 Microsoft 的 ASP.NET XML Web Service 技术或者 Microsoft 的 Windows Communication Foundation(WCF)技术，可以非常简单地处理通过 Internet 发送的消息。

- **基于 HTML 的 Web 窗体/MVC 应用程序(网站)**
  通常，ASP.NET 应用程序查询数据库并调用 Web 服务，合并和筛选返回的信息，然后使用基于 HTML 的“富”用户界面，在浏览器中显示那些信息。

- **“富”Windows GUI 应用程序**
  也可以不用网页创建 UI，而是用 Windows Store、Windows Presentation Foundation(WPF)或者 Windows Forms 技术提供的更强大、性能更好的功能。GUI 应用程序可以利用控件、菜单以及触摸/鼠标/手写笔/键盘事件，而且可以直接与底层操作系统交换信息。“富”Windows 应用程序同样可以查询数据库和使用 Web 服务。

- **Windows 控制台应用程序**
  如果对 UI 的要求很简单，那么控制台应用程序提供了一种快速、简单的方式来生成应用程序。编译器、实用程序和工具一般都是作为控制台应用程序实现的。

- **Windows 服务**
  是的，完全可以用.NET Framework 生成“服务”应用程序。通过“Windows 服务控制管理器”(Service Control Manager，SCM)控制这些服务。

- **数据库存储过程**
  Microsoft 的 SQL Server、IBM 的 DB2 以及 Oracle 的数据库服务器允许开发人员用.NET Framework 写存储过程。

- **组件库**
  .NET Framework 允许生成独立程序集(组件)，其中包含的类型可以轻松集成到前面提到的任何一种类型的应用程序中。



重要提示 Visual Studio 允许创建“可移植类库”项目。这种项目创建的程序集能用于多种应用程序类型，包括.NET Framework，Silverlight，Windows Phone，Windows Store 应用和 Xbox 360。

由于 FCL 包含的类型数量实在太多，所以有必要将相关的类型放到单独的命名空间。例如，**System** 命名空间(应当是你最熟悉的)包含 **Object** 基类型，其他所有类型最终都从这个基类型派生。此外，**System** 命名空间包含用于整数、字符、字符串、异常处理以及控制台 I/O 的类型。还包含一系列实用工具类型，能在不同数据类型之间进行安全转换、格式化数据类型、生成随机数和执行各种数学运算。所有应用程序都要使用来自 **System** 命名空间的类型。

为了使用 Framework 的任何功能，必须知道这个功能由什么类型提供，以及该类型包含在哪个命名空间中。许多类型都允许自定义其行为，你只需从所需的 FCL 类型派生出自己的类型，再进行自定义即可。.NET Framework 平台本质上是面向对象的，这为软件开发人员提供了一致性的编程模式。此外，开发人员可轻松创建自己的命名空间来包含自己的类型。这些命名空间和类型无缝合并到编程模式中。相较于 Win32 编程模式，这种新方式极大地简化了软件开发。

FCL 的大多数命名空间都提供了各种应用程序通用的类型。表 1-3 总结了部分常规命名空间，并简要描述了其中的类型的用途。这里列出的只是全部可用命名空间的极小部分。请参考文档来熟悉 Microsoft 发布的命名空间(它们的数量正在变得越来越多)。

表 1-3　部分常规的 FCL 命名空间

命名空间	内容说明
System	包含每个应用程序都要用到的所有基本类型
System.Data	包含用于和数据库通信以及处理数据的类型
System.IO	包含用于执行流 I/O 以及浏览目录/文件的类型
System.Net	包含进行低级网络通信，并与一些常用 Internet 协议协作的类型
System.Runtime.InteropServices	包含允许托管代码访问非托管操作系统平台功能(比如 COM 组件以及 Win32 或定制 DLL 中的函数)的类型
System.Security	包含用于保护数据和资源的类型
System.Text	包含处理各种编码(比如 ASCII 和 Unicode)文本的类型
System.Threading	包含用于异步操作和同步资源访问的类型
System.Xml	包含用于处理 XML 架构(XML Schema)和数据的类型

本书重点在于 CLR 以及和 CLR 密切交互的常规类型。所以，任何开发人员只要开发的应用程序或组件是面向 CLR 的，就适合阅读本书。还有其他许多不错的参考书描述了具体应用程序类型，包括 Web Services、Web 窗体/MVC 和 Windows Presentation Foundation 等。这些书能指导你快速开始构建自己的应用程序。我认为这些针对具体应用程序的参考书有助于进行“自上而下”的学习，因为它们将重点放在具体应用程序类型上，而非放在开发平台上。相反，本书提供的信息有助于你进行“自下而上”的学习。通过阅读本书以及一本针对具体应用程序的参考书，任何类型的应用程序的开发都难不倒你了。

1.7 通用类型系统

CLR 一切都围绕类型展开。到目前为止，这一点应该很清楚了。类型向应用程序和其他类型公开了功能。通过类型，用一种编程语言写的代码能与用另一种编程语言写的代码沟通。由于类型是CLR的根本，所以Microsoft制定了一个正式的规范来描述类型的定义和行为，这就是“通用类型系统”(Common Type System，CTS)。



> 注意 Microsoft 事实上已将 CTS 和.NET Framework 的其他组件——包括文件格式、元数据、中间语言以及对底层平台的访问(P/Invoke)——提交给 ECMA 以完成标准化工作。最后形成的标准称为“公共语言基础结构”(Common Language Infrastructure，CLI)。除此之外，Microsoft 还提交了 Framework 类库的一部分、C#编程语言(ECMA-334)以及 C++/CLI 编程语言。要详细了解这些工业标准，请访问 ECMA 的 Technical Committee 39 专题网站：*http://www.ecma-international.org*。还可访问 Microsoft 自己的专题网站：*http://msdn.microsoft.com/ en-us/netframework/aa569283.aspx*。此外，Microsoft 还就 ECMA-334 和 ECMA-335 规范做出社区承诺(Community Promise)，详情请访问 *http://www.microsoft.com/openspecifications/en/us/programs/community-promise/default.aspx*。

CTS 规范规定，一个类型可以包含零个或者多个成员。本书第 II 部分“设计类型”将更详细地讨论这些成员。目前只是简单地介绍一下它们。

- **字段(Field)**
  作为对象状态一部分的数据变量。字段根据名称和类型来区分。

- **方法(Method)**
  针对对象执行操作的函数，通常会改变对象状态。方法有一个名称、一个签名以及一个或多个修饰符。签名指定参数数量(及其顺序)；参数类型；方法是否有返回值；如果有返回值，还要指定返回值类型。

- **属性(Property)**
  对于调用者，属性看起来像是字段。但对于类型的实现者，属性看起来像是一个方法(或者两个方法[①])。属性允许在访问值之前校验输入参数和对象状态，以及/或者仅在必要时才计算某个值。属性还允许类型的用户采用简化的语法。最后，属性允许创建只读或只写的“字段”。

① 称为 getter 和 setter，或者称为取值方法和赋值方法。——译注

- 事件(Event)
  事件在对象以及其他相关对象之间实现了通知机制。例如，利用按钮提供的一个事件，可在按钮被单击之后通知其他对象。

CTS 还指定了类型可见性规则以及类型成员的访问规则。例如，如果将类型标记为 **public**(在 C#中使用 **public** 修饰符)，任何程序集都能看见并访问该类型。但是，如果标记为 **assembly**(在 C#中使用 **internal** 修饰符)，只有同一个程序集中的代码才能看见并访问该类型。所以，利用 CTS 制定的规则，程序集为一个类型建立了可视边界，CLR 则强制(贯彻)了这些规则。

调用者虽然能“看见”一个类型，但并不是说就能随心所欲地访问它的成员。可利用以下选项进一步限制调用者对类型中的成员的访问。

- private
  成员只能由同一个类(class)类型中的其他成员访问。

- family
  成员可由派生类型访问，不管那些类型是否在同一个程序集中。注意，许多语言(比如 C++和 C#)都用 **protected** 修饰符来标识 **family**。

- family and assembly
  成员可由派生类型访问，但这些派生类型必须在同一个程序集中定义。许多语言(比如 C#和 Visual Basic)都没有提供这种访问控制。当然，IL 汇编语言不在此列。

- assembly
  成员可由同一个程序集中的任何代码访问。许多语言都用 **internal** 修饰符来标识 **assembly**。

- family or assembly
  成员可由任何程序集中的派生类型访问。成员也可由同一个程序集中的任何类型访问。C#用 **protected internal** 修饰符标识 **family or assembly**。

- public
  成员可由任何程序集中的任何代码访问。

除此之外，CTS 还为类型继承、虚方法、对象生存期等定义了相应的规则。这些规则在设计之初，便顺应了可以用现代编程语言来表示的语义。事实上，根本不需要专门学习 CTS 规则本身，因为你选择的语言会采用你熟悉的方式公开它自己的语言语法与类型规则。通过编译来生成程序集时，它会将语言特有的语法映射到 IL——也就是 CLR 的“语言”。

接触 CLR 后不久，我便意识到最好区别对待“代码的语言”和“代码的行为”。使用 C++/CLI 可以定义自己的类型，这些类型有它们自己的成员。当然，也可使用 C#或 Visual Basic 来定义相同的类型，并在其中添加相同的成员。使用的语言不同，用于定义类型的语法也不同。但是，无论使用哪一种语言，类型的行为都完全一致，因为最终是由 CLR 的 CTS 来

定义类型的行为。

为了更形象地理解这一点，让我们来举一个例子。CTS 规定一个类型只能从一个基类派生(单继承)。因此，虽然 C++语言允许一个类型继承自多个基类型(多继承)，但 CTS 既不能接受、也不能操作这样的类型。为了帮助开发人员，Microsoft 的 C++/CLI 编译器一旦检测到你试图创建的托管代码含有从多个基类型派生的类型，就会报错。

下面是另一条 CTS 规则：所有类型最终必须从预定义的 **System.Object** 类型继承。可以看出，**Object** 是 **System** 命名空间中定义的一个类型的名称。**Object** 是其他所有类型的根，因而保证了每个类型实例都有一组最基本的行为。具体地讲，**System.Object** 类型允许做下面这些事情。

- 比较两个实例的相等性。
- 获取实例的哈希码。
- 查询一个实例的真正类型。
- 执行实例的浅(按位)拷贝。
- 获取实例对象当前状态的字符串表示。

1.8 公共语言规范

不同语言创建的对象可通过 COM 相互通信。CLR 则集成了所有语言，用一种语言创建的对象在另一种语言中，和用后者创建的对象具有相同地位。之所以能实现这样的集成，是因为 CLR 使用了标准类型集、元数据(自描述的类型信息)以及公共执行环境。

语言集成是一个宏伟的目标，最棘手的问题是各种编程语言存在极大区别。例如，有的语言不区分大小写，有的不支持 **unsigned**(无符号)整数、操作符重载或者参数数量可变的方法。

要创建很容易从其他编程语言中访问的类型，只能从自己的语言中挑选其他所有语言都支持的功能。为了在这个方面提供帮助，Microsoft 定义了“公共语言规范”(Common Language Specification，CLS)，它详细定义了一个最小功能集。任何编译器只有支持这个功能集，生成的类型才能兼容由其他符合 CLS、面向 CLR 的语言生成的组件。

CLR/CTS 支持的功能比 CLS 定义的多得多，CLS 定义的只是一个子集。所以，如果不关心语言之间的互操作性，可以开发一套功能很全的类型，它们仅受你选择的那种语言的功能集的限制。具体地说，在开发类型和方法时，如果希望它们对外“可见”，能从符合 CLS 的任何编程语言中访问，就必须遵守 CLS 定义的规则。注意，假如代码只是从定义(这些代码的)程序集的内部访问，CLS 规则就不适用了。图 1-6 形象地演示了这一段想要表达的意思。

如图 1-6 所示，CLR/CTS 提供了一个功能集。有的语言公开了 CLR/CTS 的一个较大的子

集。如果开发人员用 IL 汇编语言写程序，可以使用 CLR/CTS 提供的全部功能。但是，其他大多数语言(比如 C#、Visual Basic 和 Fortran)只向开发人员公开了 CLR/CTS 的一个功能子集。CLS 定义了所有语言都必须支持的最小功能集。

用一种语言定义类型时，如果希望在另一种语言中使用该类型，就不要在该类型的 **public** 和 **protected** 成员中使用位于 CLS 外部的任何功能。否则，其他开发人员使用其他语言写代码时，就可能无法访问这个类型的成员。

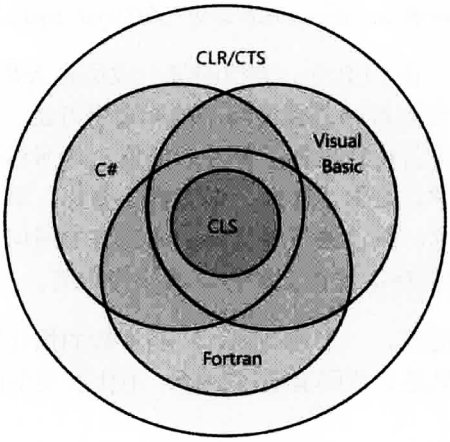


图 1-6　每种语言都提供了 CLR/CTS 的一个子集以及 CLS 的一个超集(但不一定是同一个超集)

以下代码使用 C#定义一个符合 CLS 的类型。然而，类型中含有几个不符合 CLS 的构造，造成 C#编译器报错：

```
using System;

// 告诉编译器检查 CLS 相容性
[assembly: CLSCompliant(true)]

namespace SomeLibrary {
   // 因为是 public 类，所以会显示警告
   public sealed class SomeLibraryType {

      // 警告：SomeLibrary.SomeLibraryType.Abc()的返回类型不符合 CLS
      public UInt32 Abc() { return 0; }

      // 警告：仅大小写不同的标识符 SomeLibrary.SomeLibraryType.abc()
      // 不符合 CLS
      public void abc() { }

      // 不显示警告：该方法是私有的
      private UInt32 ABC() { return 0; }
```

上述代码将**[assembly:CLSCompliant(true)]**这个特性[①]应用于程序集，告诉编译器检查其中的任何公开类型，判断是否存在任何不合适的构造阻止了从其他编程语言中访问该类型。上述代码编译时，C#编译器会报告两条警告消息。第一个警告是因为 **Abc** 方法返回无符号

① 本书按照文档将 attribute 翻译成“特性”。——译注

整数，一些语言是不能操作无符号整数值的。第二个警告是因为该类型公开了两个 **public** 方法，这两个方法(**Abc** 和 **abc**)只是大小写和返回类型有别。Visual Basic 和其他一些语言无法区分这两个方法。

有趣的是，删除 **sealed class SomeLibraryType** 之前的 **public** 字样，然后重新编译，两个警告都会消失。因为这样一来，**SomeLibraryType** 类型将默认为 **internal**(而不是 **public**)，将不再向程序集的外部公开。要获得完整的 CLS 规则列表，请参考文档的“跨语言互操作性”一节(*http://msdn.microsoft.com/zh-cn/library/730f1wy3.aspx*)。

现在提炼一下 CLS 的规则。在 CLR 中，类型的每个成员要么是字段(数据)，要么是方法(行为)。这意味着每一种编程语言都必须能访问字段和调用方法。字段和方法以特殊或通用的方式使用。为简化编程，语言往往提供了额外的抽象，从而对这些常见的编程模式进行简化。例如，语言会公开枚举、数组、属性、索引器、委托、事件、构造器、终结器、操作符重载、转换操作符等概念。编译器在源代码中遇到其中任何一样，都必须将其转换成字段和方法，使 CLR 和其他任何编程语言能够访问这些构造。

以下类型定义包含一个构造器、一个终结器、一些重载的操作符、一个属性、一个索引器和一个事件。注意，目的只是让代码能通过编译，不代表类型的正确实现方式。

```
using System;

internal sealed class Test {
   // 构造器
   public Test() {}

   // 终结器
   ~Test() {}

   // 操作符重载
   public static Boolean operator == (Test t1, Test t2) {
      return true;
   }
   public static Boolean operator != (Test t1, Test t2) {
      return false;
   }

   // 操作符重载
   public static Test operator + (Test t1, Test t2) { return null; }

   // 属性
   public String AProperty {
      get { return null; }
      set { }
   }

   // 索引器
   public String this[Int32 x] {
      get { return null; }
      set { }
   }

   // 事件
   event EventHandler AnEvent;
}
```

编译上述代码得到含有大量字段和方法的一个类型。可用.NET Framework SDK 提供的 IL

反汇编器工具(ILDasm.exe)检查最终生成的托管模块，如图 1-7 所示。

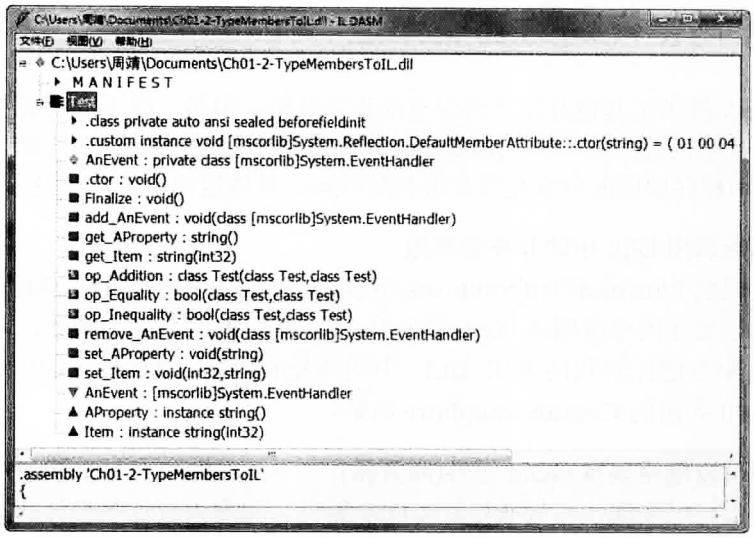


图 1-7　ILDasm 显示了 Test 类型的字段和方法(从元数据中获取)

表 1-4 总结了编程语言的各种构造与 CLR 字段/方法的对应关系。

表 1-4　Test 类型的字段和方法(从元数据中获取)

类型的成员	成员的类型	对应的编程语言构造
AnEvent	字段	事件；字段名是 **AnEvent**，类型是 **System.EventHandler**
.ctor	方法	构造器
Finalize	方法	终结器
add_AnEvent	方法	事件的 add 访问器方法
get_AProperty	方法	属性的 get 访问器方法
get_Item	方法	索引器的 get 访问器方法
op_Addition	方法	+操作符
op_Equality	方法	==操作符
op_Inequality	方法	!=操作符
remove_AnEvent	方法	事件的 remove 访问器方法
set_AProperty	方法	属性的 set 访问器方法
set_Item	方法	索引器的 set 访问器方法

Test 类型还有另一些节点未在表 1-4 中列出，包括**.class**，**.custom**，**AnEvent**，**AProperty** 以及 **Item**——它们标识了类型的其他元数据。这些节点不映射到字段或方法，只是提供了类型的一些额外信息，供 CLR、编程语言或者工具访问。例如，工具可以检测到 **Test** 类型提供了一个名为 **AnEvent** 的事件，该事件借由两个方法(**add_AnEvent** 和 **remove_AnEvent**)公开。

1.9 与非托管代码的互操作性

.NET Framework 提供了其他开发平台没有的许多优势。但是，能下定决心重新设计和重新实现全部现有代码的公司并不多。Microsoft 也知道这个问题，并通过 CLR 来提供了一些机制，允许在应用程序中同时包含托管和非托管代码。具体地说，CLR 支持三种互操作情形。

- **托管代码能调用 DLL 中的非托管函数**
  托管代码通过 P/Invoke(Platform Invoke)机制调用 DLL 中的函数。毕竟，FCL 中定义的许多类型都要在内部调用从 Kernel32.dll、User32.dll 等导出的函数。许多编程语言都提供了机制方便托管代码调用 DLL 中的非托管函数。例如，C#应用程序可调用从 Kernel32.dll 导出的 **CreateSemaphore** 函数。

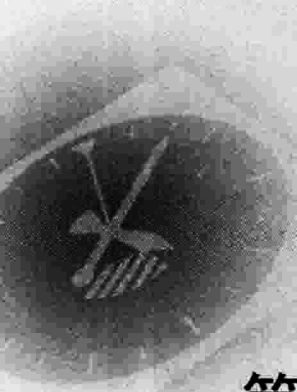
- **托管代码可以使用现有 COM 组件(服务器)**
  许多公司都已经实现了大量非托管 COM 组件。利用来自这些组件的类型库，可创建一个托管程序集来描述 COM 组件。托管代码可像访问其他任何托管类型一样访问托管程序集中的类型。这方面的详情可以参考.NET Framework SDK 提供的 TlbImp.exe 工具。有时可能没有类型库，或者想对 TlbImp.exe 生成的内容进行更多控制。这时可在源代码中手动构建一个类型，使 CLR 能用它实现正确的互操作性，例如可从 C#应用程序中使用 DirectX COM 组件。

- **非托管代码可以使用托管类型(服务器)**
  许多现有的非托管代码要求提供 COM 组件来确保代码正确工作。使用托管代码可以更简单地实现这些组件，避免所有代码都不得不和引用计数以及接口打交道。例如，可用 C#创建 ActiveX 控件或 shell 扩展。这方面的详情可以参考.NET Framework SDK 提供的 TlbExp.exe 和 RegAsm.exe 工具。



> **注意** 为了方便需要与本机代码交互的开发人员，Microsoft 公开了 Type Library Importer 工具和 P/Invoke Interop Assistant 工具的源代码。访问 *http://CLRInterop.CodePlex.com* 下载这些工具及其源代码。

Microsoft 随同 Windows 8 引入了称为 Windows Runtime(WinRT)的新 Windows API。该 API 内部通过 COM 组件来实现。但不是使用类型库文件，COM 组件是通过.NET Framework 团队创建的元数据 ECMA 标准描述其 API。好处是用一种.NET 语言写的代码(在很大程度上)能与 WinRT API 无缝对接。CLR 在幕后执行需要的所有 COM 互操作，不要求你使用任何额外的工具。第 25 章“与 WinRT 组件互操作”将进一步详细讲解。



第 2 章　生成、打包、部署和管理应用程序及类型

本章内容：

- .NET Framework 部署目标
- 将类型生成到模块中
- 元数据概述
- 将模块合并成程序集
- 程序集版本资源信息
- 语言文化
- 简单应用程序部署(私有部署的程序集)
- 简单管理控制(配置)

在解释如何为 Microsoft .NET Framework 开发程序之前，首先讨论一下生成、打包和部署应用程序及其类型的步骤。本章重点解释如何生成仅供自己的应用程序使用的程序集。第 3 章“共享程序集和强命名程序集”将讨论更高级的概念，包括如何生成和使用程序集，使其中包含的类型能由多个应用程序共享。这两章会谈及管理员能采取什么方式来影响应用程序及其类型的执行。

当今的应用程序都由多个类型构成，这些类型通常是由你和 Microsoft 创建的。除此之外，作为一个新兴产业，组件厂商们也纷纷着手构建一些专用类型，并将其出售给各大公司，以缩短软件项目的开发时间。开发这些类型时，如果使用的语言是面向 CLR 的，这些类型就能无缝地共同工作。换言之，用一种语言写的类型可以将另一个类型作为自己的基类使用，不用关心基类用什么语言开发。

本章将解释如何生成这些类型，并将其打包到文件中以进行部署。另外，还会提供一个简短的历史回顾，帮助开发人员理解.NET Framework 希望解决的某些问题。

2.1　.NET Framework 部署目标

Windows 多年来一直因为不稳定和过于复杂而口碑不佳。不管对它的评价对不对，之所以造成这种状况，要归咎于几方面的原因。首先，所有应用程序都使用来自 Microsoft 或其他

厂商的动态链接库(Dynamic-Link Library，DLL)。由于应用程序要执行多个厂商的代码，所以任何一段代码的开发人员都不能百分之百保证别人以什么方式使用这段代码。虽然这种交互可能造成各种各样的麻烦，但实际一般不会出太大的问题，因为应用程序在部署前会进行严格测试和调试。

但对于用户，当一家公司决定更新其软件产品的代码，并将新文件发送给他们时，就可能出问题。新文件理论上应该向后兼容以前的文件，但谁能对此保证呢？事实上，一家厂商更新代码时，经常都不可能重新测试和调试之前发布的所有应用程序，无法保证自己的更改不会造成不希望的结果。

很多人都可能遭遇过这样的问题：安装新应用程序时，它可能莫名其妙破坏了另一个已经安装好的应用程序。这就是所谓的“DLL hell”。这种不稳定会对普通计算机用户带来不小的困扰。最终结果是用户必须慎重考虑是否安装新软件。就我个人来说，有一些重要的应用程序是平时经常都要用到的。为了避免对它们产生不好的影响，我不会冒险去“尝鲜”。

造成 Windows 口碑不佳的第二个原因是安装的复杂性。如今，大多数应用程序在安装时都会影响到系统的全部组件。例如，安装一个应用程序会将文件复制到多个目录，更新注册表设置，并在桌面和“开始”菜单上安装快捷方式。问题是，应用程序不是一个孤立的实体。应用程序备份不易，因为必须复制应用程序的全部文件以及注册表中的相关部分。除此之外，也不能轻松地将应用程序从一台机器移动到另一台机器。只有再次运行安装程序，才能确保所有文件和注册表设置的正确性。最后，即使卸载或移除了应用程序，也免不了担心它的一部分内容仍潜伏在我们的机器中。

第三个原因涉及安全性。应用程序安装时会带来各种文件，其中许多是由不同的公司开发的。此外，Web 应用程序经常会悄悄下载一些代码(比如 ActiveX 控件)，用户根本注意不到自己的机器上安装了这些代码。如今，这种代码能够执行任何操作，包括删除文件或者发送电子邮件。用户完全有理由害怕安装新的应用程序，因为它们可能造成各种各样的危害。考虑到用户的感受，安全性必须集成到系统中，使用户能够明确允许或禁止各个公司开发的代码访问自己的系统资源。

阅读本章和下一章可以知道，.NET Framework 正在尝试彻底解决 DLL hell 的问题。另外，.NET Framework 还在很大程度上解决了应用程序状态在用户硬盘中四处分散的问题。例如，和 COM 不同，类型不再需要注册表中的设置。但遗憾的是，应用程序还是需要快捷方式。安全性方面，.NET Framework 包含称为“代码访问安全性”(Code Access Security)的安全模型。Windows 安全性基于用户身份，而代码访问安全性允许宿主设置权限，控制加载的组件能做的事情。像 Microsoft SQL Server 这样的宿主应用程序只能将少许权限授予代码，而本地安装的(自寄宿)应用程序可获得完全信任(全部权限)。以后会讲到，.NET Framework 允许用户灵活地控制哪些东西能够安装，哪些东西能够运行。他们对自己机器的控制上升到一个前所未有的高度。

2.2　将类型生成到模块中

本节讨论如何将包含多个类型的源代码文件转变为可以部署的文件。先看下面这个简单的应用程序：

```
public sealed class Program {
   public static void Main() {
      System.Console.WriteLine("Hi");
   }
}
```

该应用程序定义了 **Program** 类型，其中有名为 **Main** 的 **public static** 方法。**Main** 中引用了另一个类型 **System.Console**。**System.Console** 是 Microsoft 实现好的类型，用于实现这个类型的各个方法的 IL 代码存储在 MSCorLib.dll 文件中。总之，应用程序定义了一个类型，还使用了其他公司提供的类型。

为了生成这个示例应用程序，请将上述代码放到一个源代码文件中(假定为 Program.cs)，然后在命令行执行以下命令：

```
csc.exe /out:Program.exe /t:exe /r:MSCorLib.dll Program.cs
```

这个命令行指示 C# 编译器生成名为 Program.exe 的可执行文件(**/out:Program.exe**)。生成的文件是 Win32 控制台应用程序类型(**/t[arget]:exe**)。

C#编译器处理源文件时，发现代码引用了 **System.Console** 类型的 **WriteLine** 方法。此时，编译器要核实该类型确实存在，它确实有 **WriteLine** 方法，而且传递的实参与方法形参匹配。由于该类型在 C#源代码中没有定义，所以要顺利通过编译，必须向 C#编译器提供一组程序集，使它能解析对外部类型的引用。在上述命令行中，我添加了 **/r[eference]:MSCorLib.dll** 开关，告诉编译器在 MSCorLib.dll 程序集中查找外部类型。

MSCorLib.dll 是特殊文件，它包含所有核心类型，包括 **Byte**，**Char**，**String**，**Int32** 等等。事实上，由于这些类型使用得如此频繁，以至于 C#编译器会自动引用 MSCorLib.dll 程序集。换言之，命令行其实可以简化成下面这样(省略/**r** 开关)：

```
csc.exe /out:Program.exe /t:exe Program.cs
```

此外，由于**/out:Program.exe** 和**/t:exe** 开关是 C#编译器的默认设定，所以能继续简化成以下形式：

```
csc.exe Program.cs
```

如果因为某个原因不想 C#编译器自动引用 MSCorLib.dll 程序集，可以使用**/nostdlib** 开关。Microsoft 生成 MSCorLib.dll 程序集自身时便使用了这个开关。例如，用以下命令行编译 Program.cs 会报错，因为它使用的 **System.Console** 类型是在 MSCorLib.dll 中定义的：

```
csc.exe /out:Program.exe /t:exe /nostdlib Program.cs
```

现在更深入地思考一下 C#编译器生成的 Program.exe 文件。这个文件到底是什么？首先，

它是标准 PE(可移植执行体，Portable Executable)文件。这意味着运行 32 位或 64 位 Windows 的计算机能加载它，并能通过它执行某些操作。Windows 支持三种应用程序。生成控制台用户界面(Console User Interface，CUI)应用程序使用**/t:exe** 开关；生成图形用户界面(Graphical User Interface，GUI)应用程序使用**/t:winexe** 开关；生成 Windows Store 应用使用**/t:appcontainerexe** 开关。

响应文件

结束对编译器开关的讨论之前，让我们花点时间了解一下**响应文件**。响应文件是包含一组编译器命令行开关的文本文件。执行 CSC.exe 时，编译器打开响应文件，并使用其中包含的所有开关，感觉就像是这些开关直接在命令行上传递给 CSC.exe。要告诉编译器使用响应文件，在命令行中，请在@符号之后指定响应文件的名称。例如，假定响应文件 MyProject.rsp 包含以下文本：

```
/out:MyProject.exe
/target:winexe
```

为了让 CSC.exe 使用这些设置，可以像下面这样调用它：

```
csc.exe @MyProject.rsp CodeFile1.cs CodeFile2.cs
```

这就告诉了 C#编译器输出文件的名称和要创建哪种类型的应用程序。可以看出，响应文件能带来一些便利，不必每次编译项目时都手动指定命令行参数。

C#编译器支持多个响应文件。除了在命令行上显式指定的文件，编译器还会自动查找名为 CSC.rsp 的文件。CSC.exe 运行时，会在 CSC.exe 所在的目录查找全局 CSC.rsp 文件。想应用于自己所有项目的设置应放到其中。编译器汇总并使用所有响应文件中的设置。本地和全局响应文件中的某个设置发生冲突，将以本地设置为准。类似地，命令行上显式指定的设置将覆盖本地响应文件中的设置。

.NET Framework 安装时会在%SystemRoot%\Microsoft.NET\Framework(*64*)\v*X.X.X* 目录中安装默认全局 CSC.rsp 文件(*X.X.X* 是你安装的.NET Framework 的版本号)。这个文件的最新版本包含以下开关：

```
# This file contains command-line options that the C#
# command line compiler (CSC) will process as part
# of every compilation, unless the "/noconfig" option
# is specified.
# Reference the common Framework libraries
/r:Accessibility.dll
/r:Microsoft.CSharp.dll
/r:System.Configuration.dll
/r:System.Configuration.Install.dll
/r:System.Core.dll
/r:System.Data.dll
/r:System.Data.DataSetExtensions.dll
/r:System.Data.Linq.dll
/r:System.Data.OracleClient.dll
/r:System.Deployment.dll
/r:System.Design.dll
/r:System.DirectoryServices.dll
/r:System.dll
```

```
/r:System.Drawing.Design.dll
/r:System.Drawing.dll
/r:System.EnterpriseServices.dll
/r:System.Management.dll
/r:System.Messaging.dll
/r:System.Runtime.Remoting.dll
/r:System.Runtime.Serialization.dll
/r:System.Runtime.Serialization.Formatters.Soap.dll
/r:System.Security.dll
/r:System.ServiceModel.dll
/r:System.ServiceModel.Web.dll
/r:System.ServiceProcess.dll
/r:System.Transactions.dll
/r:System.Web.dll
/r:System.Web.Extensions.Design.dll
/r:System.Web.Extensions.dll
/r:System.Web.Mobile.dll
/r:System.Web.RegularExpressions.dll
/r:System.Web.Services.dll
/r:System.Windows.Forms.Dll
/r:System.Workflow.Activities.dll
/r:System.Workflow.ComponentModel.dll
/r:System.Workflow.Runtime.dll
/r:System.Xml.dll
/r:System.Xml.Linq.dll
```

由于全局 CSC.rsp 文件引用了列出的所有程序集，所以不必使用 C#编译器的**/reference** 开关显式引用这些程序集。这个响应文件为开发人员带来了极大的方便，因为可以直接使用 Microsoft 发布的各个程序集中定义的类型和命名空间，不必每次编译时都指定**/reference** 编译器开关。

引用所有这些程序集对编译器的速度有一点影响。但是，如果源代码没有引用上述任何程序集定义的类型或成员，就不会影响最终的程序集文件，也不会影响程序的执行性能。

当然，要进一步简化操作，还可在全局 CSC.rsp 文件中添加自己的开关。但这样一来，在其他机器上重现代码的生成环境就比较困难了：在每台用于生成的机器上，都必须以相同方式更新 CSC.rsp。另外，指定**/noconfig** 命令行开关，编译器将忽略本地和全局 CSC.rsp 文件。

2.3　元数据概述

现在，我们知道了创建的是什么类型的 PE 文件。但是，Program.exe 文件中到底有什么？托管 PE 文件由 4 部分构成：PE32(+)头、CLR 头、元数据以及 IL。PE32(+)头是 Windows 要求的标准信息。CLR 头是一个小的信息块，是需要 CLR 的模块(托管模块)特有的。这个头包含模块生成时所面向的 CLR 的 major(主)和 minor(次)版本号；一些标志(flag)；一个 **MethodDef** token(稍后详述)，该 token 指定了模块的入口方法(前提是该模块是 CUI、GUI 或 Windows Store 执行体)；一个可选的强名称数字签名(将在第 3 章讨论)。最后，CLR 头还包含模块内部的一些元数据表的大小和偏移量。可以查看 CorHdr.h 头文件定义的 **IMAGE_COR20_HEADER** 来了解 CLR 头的具体格式。

元数据是由几个表构成的二进制数据块。有三种表，分别是定义表(definition table)、引用

表(reference table)和清单表(manifest table)。表 2-1 总结了模块元数据块中常用的定义表。

表 2-1 常用的元数据定义表

元数据定义表名称	说明
ModuleDef	总是包含对模块进行标识的一个记录项。该记录项包含模块文件名和扩展名(不含路径)，以及模块版本 ID(形式为编译器创建的 GUID)。这样可在保留原始名称记录的前提下自由重命名文件。但强烈反对重命名文件，因为可能妨碍 CLR 在运行时正确定位程序集
TypeDef	模块定义的每个类型在这个表中都有一个记录项。每个记录项都包含类型的名称、基类型、一些标志(**public**，**private** 等)以及一些索引，这些索引指向 MethodDef 表中该类型的方法、FieldDef 表中该类型的字段、PropertyDef 表中该类型的属性以及 EventDef 表中该类型的事件
MethodDef	模块定义的每个方法在这个表中都有一个记录项。每个记录项都包含方法的名称、一些标志(**private**，**public**，**virtual**，**abstract**，**static**，**final** 等)、签名以及方法的 IL 代码在模块中的偏移量。每个记录项还引用了 ParamDef 表中的一个记录项，后者包括与方法参数有关的更多信息
FieldDef	模块定义的每个字段在这个表中都有一个记录项。每个记录项都包含标志(**private**，**public** 等)、类型和名称
ParamDef	模块定义的每个参数在这个表中都有一个记录项。每个记录项都包含标志(**in**，**out**，**retval** 等)、类型和名称
PropertyDef	模块定义的每个属性在这个表中都有一个记录项。每个记录项都包含标志、类型和名称
EventDef	模块定义的每个事件在这个表中都有一个记录项。每个记录项都包含标志和名称

编译器编译源代码时，代码定义的任何东西都导致在表 2-1 列出的某个表中创建一个记录项。此外，编译器还会检测源代码引用的类型、字段、方法、属性和事件，并创建相应的元数据表记录项。在创建的元数据中包含一组引用表，它们记录了所引用的内容。表 2-2 总结了常用的引用元数据表。

表 2-2 常用的引用元数据表

引用元数据表名称	说明
AssemblyRef	模块引用的每个程序集在这个表中都有一个记录项。每个记录项都包含绑定[①]该程序集所需的信息：程序集名称(不含路径和扩展名)、版本号、语言文化(culture)以及公钥 token(根据发布者的公钥生成的一个小的哈希值，标识了所引用程序集的发布者)。每个记录项还包含一些标志(flag)和一个哈希值。该哈希值本应作为所引用程序集的二进制数据的校验和来使用。但是，目前 CLR 完全忽略该哈希值，未来的 CLR 可能同样如此

① bind 在文档中有时翻译成“联编”，binder 有时翻译成“联编程序”。本书采用“绑定”和“绑定器”。——译注

续表

引用元数据表名称	说明
ModuleRef	实现该模块所引用的类型的每个 PE 模块在这个表中都有一个记录项。每个记录项都包含模块的文件名和扩展名(不含路径)。可能是别的模块实现了你需要的类型，这个表的作用便是建立同那些类型的绑定关系
TypeRef	模块引用的每个类型在这个表中都有一个记录项。每个记录项都包含类型的名称和一个引用(指向类型的位置)。如果类型在另一个类型中实现，引用指向一个 TypeRef 记录项。如果类型在同一个模块中实现，引用指向一个 ModuleDef 记录项。如果类型在调用程序集内的另一个模块中实现，引用指向一个 ModuleRef 记录项。如果类型在不同的程序集中实现，引用指向一个 AssemblyRef 记录项
MemberRef	模块引用的每个成员(字段和方法，以及属性方法和事件方法)在这个表中都有一个记录项。每个记录项都包含成员的名称和签名，并指向对成员进行定义的那个类型的 TypeRef 记录项

除了表 2-1 和表 2-2 所列的，还有其他许多定义表和引用表。但是，我的目的只是让你体会一下编译器在生成的元数据中添加的各种信息。前面提到还有清单(manifest)元数据表，它们将于本章稍后讨论。

可用多种工具检查托管 PE 文件中的元数据。我个人喜欢使用 ILDasm.exe，即 IL Disassembler(IL 反汇编器)。要查看元数据表，请执行以下命令行：

```
ILDasm Program.exe
```

ILDasm.exe 将运行并加载 Program.exe 程序集。要采用一种美观的、容易阅读的方式查看元数据，请选择“视图”|“元信息”|“显示!”菜单项(或直接按 Ctrl+M 组合键)。随后会显示以下信息：

```
===========================================================
ScopeName     : Program.exe
MVID          : {CA73FFE8-0D42-4610-A8D3-9276195C35AA}
===========================================================
Global functions
-----------------------------------------------------------

Global fields
-----------------------------------------------------------

Global MemberRefs
-----------------------------------------------------------

TypeDef #1 (02000002)
-----------------------------------------------------------
    TypDefName    : Program  (02000002)
    Flags         : [Public] [AutoLayout] [Class] [Sealed] [AnsiClass] [BeforeFieldInit]  (00100101)
    Extends       : 01000001 [TypeRef] System.Object
    Method #1 (06000001) [ENTRYPOINT]
    -------------------------------------------------------
        MethodName    : Main (06000001)
        Flags         : [Public] [Static] [HideBySig] [ReuseSlot]  (00000096)
        RVA           : 0x00002050
        ImplFlags     : [IL] [Managed]  (00000000)
        CallCnvntn    : [DEFAULT]
```

```
        ReturnType    : Void
        No arguments.

    Method #2 (06000002)
    -----------------------------------------------------
        MethodName    : .ctor (06000002)
        Flags         : [Public] [HideBySig] [ReuseSlot] [SpecialName] [RTSpecialName]
                        [.ctor]  (00001886)
        RVA           : 0x0000205e
        ImplFlags     : [IL] [Managed]  (00000000)
        CallCnvntn    : [DEFAULT]
        hasThis
        ReturnType    : Void
        No arguments.

TypeRef #1 (01000001)
-----------------------------------------------------
Token:                0x01000001
ResolutionScope:      0x23000001
TypeRefName:          System.Object
    MemberRef #1      (0a000004)
    -------------------------------------------------
        Member: (0a000004) .ctor:
        CallCnvntn: [DEFAULT]
        hasThis
        ReturnType: Void
        No arguments.

TypeRef #2 (01000002)
-----------------------------------------------------
Token:                0x01000002
ResolutionScope:      0x23000001
TypeRefName:          System.Runtime.CompilerServices.CompilationRelaxationsAttribute
    MemberRef #1      (0a000001)
    -------------------------------------------------
        Member: (0a000001) .ctor:
        CallCnvntn: [DEFAULT]
        hasThis
        ReturnType: Void
        1 Arguments
            Argument #1:  I4

TypeRef #3 (01000003)
-----------------------------------------------------
Token:                0x01000003
ResolutionScope:      0x23000001
TypeRefName:          System.Runtime.CompilerServices.RuntimeCompatibilityAttribute
    MemberRef #1      (0a000002)
    -------------------------------------------------
        Member: (0a000002) .ctor:
        CallCnvntn: [DEFAULT]
        hasThis
        ReturnType: Void
        No arguments.

TypeRef #4 (01000004)
-----------------------------------------------------
Token:                0x01000004
ResolutionScope:      0x23000001
TypeRefName:          System.Console
    MemberRef #1      (0a000003)
    -------------------------------------------------
        Member: (0a000003) WriteLine:
        CallCnvntn: [DEFAULT]
        ReturnType: Void
        1 Arguments
            Argument #1:  String
```

```
Assembly
-------------------------------------------------------
    Token: 0x20000001
    Name : Program
    Public Key    :
    Hash Algorithm : 0x00008004
    Version: 0.0.0.0
    Major Version: 0x00000000
    Minor Version: 0x00000000
    Build Number: 0x00000000
    Revision Number: 0x00000000
    Locale: <null>
    Flags : [none] (00000000)
    CustomAttribute #1 (0c000001)
    -----------------------------------------------------
        CustomAttribute Type: 0a000001
        CustomAttributeName:
        System.Runtime.CompilerServices.CompilationRelaxationsAttribute ::
           instance void .ctor(int32)
        Length: 8
        Value : 01 00 08 00 00 00 00 00                    >            <
        ctor args: (8)

    CustomAttribute #2 (0c000002)
    -----------------------------------------------------
        CustomAttribute Type: 0a000002
        CustomAttributeName:
        System.Runtime.CompilerServices.RuntimeCompatibilityAttribute ::
           instance void .ctor()
        Length: 30
        Value : 01 00 01 00 54 02 16 57  72 61 70 4e 6f 6e 45 78 >    T  WrapNonEx<
              : 63 65 70 74 69 6f 6e 54  68 72 6f 77 73 01      >ceptionThrows  <
        ctor args: ()

AssemblyRef #1 (23000001)
-------------------------------------------------------
    Token: 0x23000001
    Public Key or Token: b7 7a 5c 56 19 34 e0 89
    Name: mscorlib
    Version: 4.0.0.0
    Major Version: 0x00000004
    Minor Version: 0x00000000
    Build Number: 0x00000000
    Revision Number: 0x00000000
    Locale: <null>
    HashValue Blob:
    Flags: [none] (00000000)

User Strings
-------------------------------------------------------
70000001 : ( 2) L"Hi"

Coff symbol name overhead:  0
=======================================================
=======================================================
=======================================================
```

幸好 ILDasm 处理了元数据表，恰当合并了信息，避免我们分析原始表。例如，可以看到当 ILDasm 显示一个 TypeDef 记录项时，会在第一个 TypeRef 项之前显示对应的成员定义信息。

不用完全理解上面显示的一切。重点是 Program.exe 包含名为 **Program** 的 TypeDef。**Program** 是公共密封类，从 **System.Object** 派生(**System.Object** 是引用的另一个程序集中的类型)。**Program** 类型还定义了两个方法：**Main** 和**.ctor**(构造器)。

Main 是公共静态方法，用 IL 代码实现(有的方法可能用本机 CPU 代码实现，比如 x86 代码)。**Main** 返回类型是 **void**，无参。构造器(名称始终是**.ctor**)是公共方法，也用 IL 代码实现。构造器返回类型是 **void**，无参，有一个 **this** 指针(指向调用方法时构造对象的内存)。

强烈建议多试验一下 ILDasm。它提供了丰富的信息。你对自己看到的东西理解得越多，对 CLR 及其功能的理解就越好。本书后面会大量地用到 ILDasm。

为增加趣味性，来看看 Program.exe 程序集的统计信息。在 ILDasm 中选择“视图”|“统计”，会显示以下信息：

```
File size : 3584
  PE header size : 512 (496 used) (14.29%)
  PE additional info : 1411 (39.37%)
  Num.of PE sections : 3
  CLR header size : 72 ( 2.01%)
  CLR meta-data size : 612 (17.08%)
  CLR additional info : 0 ( 0.00%)
  CLR method headers : 2 ( 0.06%)
  Managed code : 20 ( 0.56%)
  Data : 2048 (57.14%)
  Unaccounted : -1093 (-30.50%)

Num.of PE sections : 3
  .text - 1024
  .rsrc - 1536
  .reloc - 512

CLR meta-data size : 612
  Module - 1 (10 bytes)
  TypeDef - 2 (28 bytes) 0 interfaces, 0 explicit layout
  TypeRef - 4 (24 bytes)
  MethodDef - 2 (28 bytes) 0 abstract, 0 native, 2 bodies
  MemberRef - 4 (24 bytes)
  CustomAttribute- 2 (12 bytes)
  Assembly - 1 (22 bytes)
  AssemblyRef - 1 (20 bytes)
  Strings - 184 bytes
  Blobs - 68 bytes
  UserStrings - 8 bytes
  Guids - 16 bytes
  Uncategorized - 168 bytes

CLR method headers : 2
  Num.of method bodies - 2
  Num.of fat headers - 0
  Num.of tiny headers - 2

Managed code : 20
  Ave method size - 10
```

从中可以看出文件大小(字节数)以及文件各部分大小(字节数和百分比)。对于这个如此小的 Program.exe 应用程序，PE 头和元数据占了相当大的比重。事实上，IL 代码只有区区 20 个字节。当然，随着应用程序规模的增大，它会重用大多数类型以及对其他类型和程序集的引用，元数据和头信息在整个文件中的比重越来越小。



注意 顺便说一下，ILDasm.exe 的一个 bug 会影响显示的文件长度。尤其不要相信 Unaccounted 信息。

2.4　将模块合并成程序集

上一节讨论的 Program.exe 并非只是含有元数据的 PE 文件，它还是**程序集**(assembly)。程序集是一个或多个类型定义文件及资源文件的集合。在程序集的所有文件中，有一个文件容纳了**清单**(manifest)。清单也是一个元数据表集合，表中主要包含作为程序集组成部分的那些文件的名称。此外，还描述了程序集的版本、语言文化、发布者、公开导出的类型以及构成程序集的所有文件。

CLR 操作的是程序集。换言之，CLR 总是首先加载包含“清单”元数据表的文件，再根据“清单”来获取程序集中的其他文件的名称。下面列出了程序集的重要特点。

- 程序集定义了可重用的类型。
- 程序集用一个版本号标记。
- 程序集可以关联安全信息。

除了包含清单元数据表的文件，程序集其他单独的文件并不具备上述特点。

类型为了顺利地进行打包、版本控制、安全保护以及使用，必须放在作为程序集一部分的模块中。程序集大多数时候只有一个文件，就像前面的 Program.exe 那样。然而，程序集还可以由多个文件构成：一些是含有元数据的 PE 文件，另一些是.gif 或.jpg 这样的资源文件。为便于理解，可将程序集视为一个逻辑 EXE 或 DLL。

Microsoft 为什么引入“程序集”的概念？这是因为使用程序集，可重用类型的逻辑表示与物理表示就可以分开。例如，程序集可能包含多个类型。可以将常用类型放到一个文件中，不常用类型放到另一个文件中。如果程序集要从 Internet 下载并部署，那么对于含有不常用类型的文件，假如客户端永远不使用那些类型，该文件就永远不会下载到客户端。例如，擅长制作 UI 控件的一家独立软件开发商(Independent Software Vendor，ISV)可选择在单独的模块中实现 Active Accessibility 类型(以满足 Microsoft 徽标认证授权要求)[①]。这样一来，只有需要额外“无障碍访问”功能的用户才需要下载该模块。

为了配置应用程序去下载程序集文件，可在应用程序配置文件中指定 **codeBase** 元素(详见第 3 章)。在 **codeBase** 元素定义的 URL 所指向的位置，可找到程序集的所有文件。试图加载程序集的一个文件时，CLR 获取 **codeBase** 元素的 URL，检查机器的下载缓存，判断文件是否存在。如果是，直接加载文件。如果不是，CLR 去 URL 指向的位置将文件下载到缓存。如果还是找不到文件，CLR 在运行时抛出 **FileNotFoundException** 异常。

---

① Microsoft Active Accessibility®是一种基于 COM 的技术，能够为应用程序和 Active Accessibility 客户端提供标准、一致的机制来交换信息。宗旨是帮助残障人士更有效地使用计算机。——译注

我想指出使用多文件程序集的三点理由。

- 不同的类型用不同的文件，使文件能以“增量”方式下载(就像前面在 Internet 下载的例子中描述的那样)。另外，将类型划分到不同的文件中，可以对购买和安装的应用程序进行部分或分批打包/部署。

- 可在程序集中添加资源或数据文件。例如，假定一个类型的作用是计算保险信息，需要访问精算表才能完成计算。在这种情况下，不必在自己的源代码中嵌入精算表。相反，可以使用一个工具(比如稍后要讨论的程序集链接器 AL.exe)，使数据文件成为程序集的一部分。顺便说一句，数据文件可为任意格式——包括文本文件、Microsoft Office Excel 电子表格文件以及 Microsoft Office Word 表格等——只要应用程序知道如何解析。

- 程序集包含的各个类型可以用不同的编程语言来实现。例如，一些类型可以用 C#实现，一些用 Visual Basic 实现，其他则用其他语言实现。编译用 C#写的类型时，编译器会生成一个模块。编译用 Visual Basic 写的类型时，编译器会生成另一个模块。然后可以用工具将所有模块合并成单个程序集。其他开发人员在使用这个程序集时，只知道这个程序集包含了一系列类型，根本不知道、也不用知道这些类型分别是用什么语言写的。顺便说一句，如果愿意，可以对每个模块都运行 ILDasm.exe，获得相应的 IL 源代码文件。然后运行 ILAsm.exe，将所有 IL 源代码文件都传给它。随后，ILAsm.exe 会生成包含全部类型的单个文件。该技术的前提是源代码编译器能生成纯 IL 代码。



重要提示 总之，程序集是进行重用、版本控制和应用安全性设置的基本单元。它允许将类型和资源文件划分到单独的文件中。这样一来，无论你自己，还是你的程序集的用户，都可以决定打包和部署哪些文件。一旦 CLR 加载含有清单的文件，就可确定在程序集的其他文件中，具体是哪一些文件包含应用程序引用的类型和资源。程序集的用户(其他开发人员)只需知道含有清单的那个文件的名称。这样一来，文件的具体划分方式在程序集的用户那里就是完全透明的。你以后可以自由更改，不会干扰应用程序的行为。

如果多个类型能共享相同的版本号和安全性设置，建议将所有这些类型放到同一个文件中，而不是分散到多个文件中，更不要分散到多个程序集中。这是出于对性能的考虑。每次加载文件或程序集，CLR 和 Windows 都要花费一定的时间来查找、加载并初始化程序集。需要加载的文件/程序集的数量越少，性能越好，因为加载较少的程序集有助于减小工作集(working set)，并缓解进程地址空间的碎片化。最后，nGen.exe 处理较大的文件时可以进行更好的优化。

生成程序集要么选择现有的 PE 文件作为“清单”的宿主，要么创建单独的 PE 文件并只在其中包含清单。表 2-3 展示了将托管模块转换成程序集的清单元数据表。

表 2-3　清单元数据表

清单元数据表名称	说明
AssemblyDef	如果模块标识的是程序集，这个元数据表就包含单一记录项来列出程序集名称(不含路径和扩展名)、版本(major，minor，build 和 revision)、语言文化(culture)、一些标志(flag)、哈希算法以及发布者公钥(可为 **null**)
FileDef	作为程序集一部分的每个 PE 文件和资源文件在这个表中都有一个记录项(清单本身所在的文件除外，该文件在 AssemblyDef 表的单一记录项中列出)。在每个记录项中，都包含文件名和扩展名(不含路径)、哈希值和一些标志(flags)。如果程序集只包含它自己的文件，FileDef 表将无记录①
ManifestResourceDef	作为程序集一部分的每个资源在这个表中都有一个记录项。记录项中包含资源名称、一些标志(如果在程序集外部可见，就为 **public**；否则为 **private**)以及 FileDef 表的一个索引(指出资源或流包含在哪个文件中)。如果资源不是独立文件(比如.jpg 或者.gif 文件)，那么资源就是包含在 PE 文件中的流。对于嵌入资源，记录项还包含一个偏移量，指出资源流在 PE 文件中的起始位置
ExportedTypesDef	从程序集的所有 PE 模块中导出的每个 public 类型在这个表中都有一个记录项。记录项中包含类型名称、FileDef 表的一个索引(指出类型由程序集的哪个文件实现)以及 TypeDef 表的一个索引。注意，为节省空间，从清单所在文件导出的类型不再重复，因为可通过元数据的 TypeDef 表获取类型信息

由于有了清单的存在，程序集的用户不必关心程序集的划分细节。另外，清单也使程序集具有了自描述性(self-describing)。另外，在包含清单的文件中，一些元数据信息描述了哪些文件是程序集的一部分。但是，那些文件本身并不包含元数据来指出它们是程序集的一部分。



注意　包含清单的程序集文件还有一个 AssemblyRef 表。程序集全部文件引用的每个程序集在这个表中都有一个记录项。这样一来，工具只需打开程序集的清单，就可知道它引用的全部程序集，而不必打开程序集的其他文件。同样地，AssemblvRef 表的存在加强了程序集的自描述性。

指定以下任何命令行开关，C#编译器都会生成程序集：**/t[arget]:exe**，**/t[arget]:winexe**，**/t[arget]: appcontainerexe**，**/t[arget]:library** 或者**/t[arget]:winmdobj**②。所有这些开关都会造成编译器生成含有清单元数据表的 PE 文件。这些开关分别生成 CUI 执行体、GUI 执行体，Windows Store 执行体、类库或者 WINMD 库。

① 所谓“如果程序集只包含它自己的文件”，是指程序集只包含它的主模块，不包含其他非主模块和资源文件。1.2 节已经说过，程序集是一个抽象概念，是一个或者多个模块文件和资源文件组成的逻辑单元，其中必定含有且只有一个后缀为.exe 或者.dll 的主模块文件。——译注

② 若使用**/t[arget]:winmdobj**，生成的.winmdobj 文件必须传给 WinMDExp.exe 工具进行处理，以便将程序集的公共 CLR 类型作为 **Windows Runtime** 类型公开。WinMDExp.exe 工具根本不会碰 IL 代码。

除了这些开关，C#编译器还支持**/t[arget]:module** 开关。这个开关指示编译器生成一个不包含清单元数据表的 PE 文件。这样生成的肯定是一个 DLL PE 文件。CLR 要想访问其中的任何类型，必须先将该文件添加到一个程序集中。使用**/t:module** 开关时，C#编译器默认为输出文件使用.netmodule 扩展名。



重要提示 遗憾的是，不能直接从 Microsoft Visual Studio 集成开发环境(IDE)中创建多文件程序集。只能用命令行工具创建多文件程序集。

可通过许多方式将模块添加到程序集。如果用 C#编译器生成含清单的 PE 文件，可以使用**/addmodule** 开关。为了理解如何生成多文件程序集，假定有两个源代码文件。

- RUT.cs，其中包含不常用类型。
- FUT.cs，其中包含常用类型。

下面将不常用类型编译到一个单独的模块。这样一来，如果程序集的用户永远不使用不常用类型，就不需要部署这个模块。

```
csc /t:module RUT.cs
```

上述命令行造成 C#编译器创建名为 RUT.netmodule 的文件。这是一个标准的 DLL PE 文件，但是，CLR 不能单独加载它。

接着将常用类型编译到另一个模块。该模块将成为程序集清单的宿主，因为这些类型会经常用到。事实上，由于该模块现在代表整个程序集，所以我将输出文件的名称改为 MultiFileLibrary.dll，而不是默认的 FUT.dll。

```
csc /out:MultiFileLibrary.dll /t:library /addmodule:RUT.netmodule FUT.cs
```

上述命令行指示 C#编译器编译 FUT.cs 来生成 MultiFileLibrary.dll。由于指定了**/t:library** 开关，所以生成的是含有清单元数据表的 DLL PE 文件。**/addmodule:RUT.netmodule** 开关告诉编译器 RUT.netmodule 文件是程序集的一部分。具体地说，**/addmodule** 开关告诉编译器将文件添加到 FileDef 清单元数据表，并将 RUT.netmodule 的公开导出类型添加到 ExportedTypesDef 清单元数据表。

编译器最终创建图 2-1 所示的两个文件。清单在右边的模块中。

RUT.netmodule 文件包含编译 RUT.cs 所生成的 IL 代码。该文件还包含一些定义元数据表，描述了 RUT.cs 定义的类型、方法、字段、属性、事件等。还包含一些引用元数据表，描述了 RUT.cs 引用的类型、方法等。MultiFileLibrary.dll 是一个单独的文件。与 RUT.netmodule 相似，MultiFileLibrary.dll 包含编译 FUT.cs 所生成的 IL 代码以及类似的定义与引用元数据表。然而，MultiFileLibrary.dll 还包含额外的清单元数据表，这使 MultiFileLibrary.dll 成为了程序集。清单元数据表描述了程序集的所有文件(MultiFileLibrary.dll 本身和 RUT.netmodule)。清单元数据表还包含从 MultiFileLibrary.dll 和 RUT.netmodule 导出的所有公共类型。

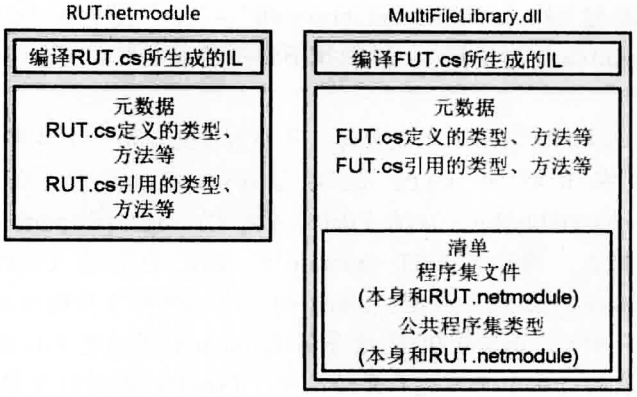


图 2-1　含有两个托管模块的多文件程序集，清单在其中一个模块中



> **注意**　清单元数据表实际并不包含从清单所在的 PE 文件导出的类型。这是一项优化措施，旨在减少 PE 文件中的清单信息量。因此，上述说法“清单元数据表还包含从 MultiFileLibrary.dll 和 RUT.netmodule 导出的所有公共类型”并非百分之百准确。不过，这种说法确实精准地反映了清单在逻辑意义上公开的内容。

生成 MultiFileLibrary.dll 程序集之后，接着可用 ILDasm.exe 检查元数据的清单表，验证程序集文件确实包含了对 RUT.netmodule 文件的类型的引用。FileDef 和 ExportedTypesDef 元数据表的内容如下所示。

```
File #1 (26000001)
-------------------------------------------------------
    Token: 0x26000001
    Name : RUT.netmodule
    HashValue Blob : e6 e6 df 62 2c a1 2c 59 97 65 0f 21 44 10 15 96 f2 7e db c2
    Flags : [ContainsMetaData]  (00000000)


ExportedType #1 (27000001)
-------------------------------------------------------
    Token: 0x27000001
    Name: ARarelyUsedType
    Implementation token: 0x26000001
    TypeDef token: 0x02000002
    Flags     : [Public] [AutoLayout] [Class] [Sealed] [AnsiClass]
                [BeforeFieldInit]  (00100101)
```

可以看出，RUT.netmodule 文件已被视为程序集的一部分，它的 token 是 0x26000001。在 ExportedTypesDef 表中可以看到一个公开导出的类型，名为 **ARarelyUsedType**。该类型的实现 token 是 0x26000001，表明类型的 IL 代码包含在 RUT.netmodule 文件中。

客户端代码必须使用 **/r[eference]:MultiFileLibrary.dll** 编译器开关生成，才能使用 MultiFileLibrary.dll 程序集的类型。该开关指示编译器在搜索外部类型时加载 MultiFileLibrary.dll 程序集以及 FileDef 表中列出的所有文件。要求程序集的所有文件都已安装，而且能够访问。删除 RUT.netmodule 文件导致 C#编译器会报告以下错误：**fatal error**